TEC-0017

An Image Understanding Environment for DARPA Supported Research and Applications, First Annual Report

Tod Levitt Scott Barclay
Scott Johnston John Dye
Advanced Decision Systems
1500 Plymouth Street
Mountain View, CA 94043-1230

Daryl Lawton
Georgia Institute of Technology
225 North Avenue, NW
Atlanta, GA 30332-0320

October 1991

Prepared for:
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22209-2308
Monitored by:
U.S. Army Corps of Engineers
Topographic Engineering Center
Fort Belvoir, Virginia 22060-5546

Destroy this report when no longer needed.
Do not return it to the originator.

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

The citation in this report of trade names of commercially available products does not constitute official endorsement or approval of the use of such products.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	October 1991	Technical Report Sep. 1989 - Sep. 1990

4. TITLE AND SUBTITLE

An Image Understanding Environment for DARPA Supported Research and Applications, First Annual Report

5. FUNDING NUMBERS

DACA76-89-C-0023

6. AUTHOR(S)

[1]Tod Levitt [1]Scott Barclay [2]Daryl Lawton
[1]Scott Johnston [1]John Dye

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

[1]Advanced Decision Systems, 1500 Plymouth Street, Mountain View, CA 94043-1230

[2]Georgia Institute of Technology, Atlanta, GA 30332-0320

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency (DARPA)
1400 Wilson Blvd., Arlington VA 22209-1155

U.S. Army Topographic Engineering Center
Fort Belvoir, VA 22060-5546

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

TEC-0017

11. SUPPLEMENTARY NOTES

Effective 1 October 1991, the U.S. Army Engineer Topographic Laboratories (ETL) became the U.S. Army Topographic Engineering Center (TEC).

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report presents the functional specifications and top-level constructs of the core design of an image understanding (IU) application development environment. The core environment presented in this report provides tools to leverage the development of IU applications and to facilitate transfer of IU and reasoning technology from its origins in research laboratories into IU applications. The core environment provides both a development platform and reusable components including a library of image processing and IU routines and data structures, and an integrated set of high-level reasoning capabilities such as bayesian networks and logic engines. The environment design is an object oriented structure built on the C++ programming language. The report also addresses system engineering issues in applying the environment to develop workstations specialized for terrain analysis and medical applications.

14. SUBJECT TERMS

Vision environment, Image Understanding (IU), Computer Vision (CV), object oriented programming

15. NUMBER OF PAGES

80

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UNLIMITED

Table of Contents

Table of Figures

PREFACE

This report was prepared under contract DACA76-89-C-0023 for the U.S. Army Topographic Engineering Center, Fort Belvoir, Virginia 22060-5546 by Advanced Decision Systems, 1500 Plymouth Street, Mountain View, CA 94043-1230, and Georgia Institute of Technology, 225 North Avenue, NW, Atlanta, GA 30332-0320. The Contracting Officer's Representative was Mr. George Lukes.

1.0 Vision Environment Report Overview

This report presents the functional specifications and top-level constructs of the core design of an image understanding (IU) application development environment. It also addresses system engineering issues in applying the environment to develop workstations specialized for terrain analysis and medical applications. The environment build has also been started this year.

In addition to designing and building this nearterm environment, Advanced Decision Systems (ADS) and Georgia Institute of Technology (GT) have actively participated in the IU community's design of a DARPA-ISTO sponsored, portable IU software environment. This environment is intended to facilitate the transfer of IU community technology into industrial, military, and commercial applications. GT and ADS headed the design committee on knowledge representation in the preliminary design effort from 5/90 through 9/90, and are currently a part of an independent design team in the continuing design effort.

The core environment presented in this report provides tools to leverage the development of IU applications and to facilitate transfer of IU and reasoning technology from its origins in research laboratories into IU applications. The core environment provides both a development platform and reusable components including a library of image processing and IU routines and data structures, and an integrated set of higher-level reasoning capabilities such as bayesian networks and logic engines. This layered software evnvironment concept is pictured in

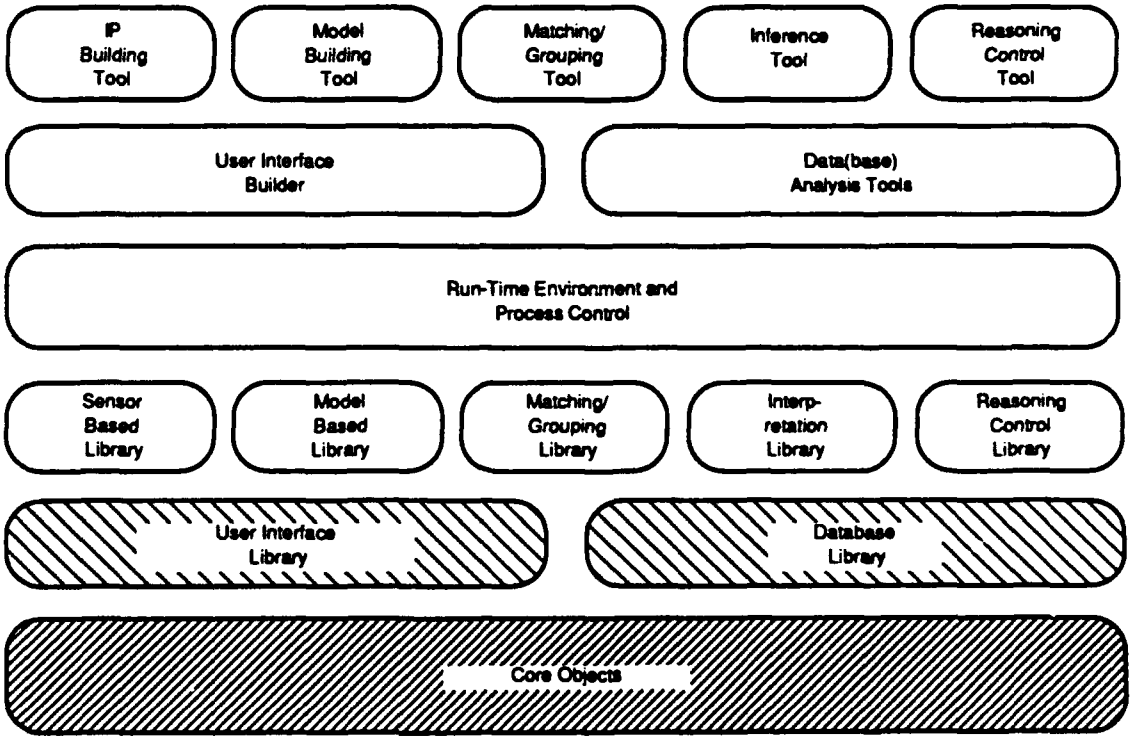


Figure 1: IU Application Development Environment Concept

This program is focused on the design and development of the core objects necessary for the foundataion of the environment. These correspond to the shaded boxes in Figure 1. The primary accomplishments of the project so far include

- Creation of a functional specification and top level design for an IU software environment integrating model-based reasoning, image processing, automated inference and hypermedia capability,

- Development of an IU environment class structure,

- Implementation of a partial object hierarchy, including build of a basic set of user interface and imagery manipulation classes, extension of image objects to include arbitrary gray level polygons, graphical interaction with remote databases, and Bayes net objects integrated with feature extraction capabilities,

- Providing a system engineering analysis of tasks and requirements for diverse IU applications and distilled a common core of IU workstation requirements, and

- Identifying and integrating key public domain software to provide image processing and user interface capabilities.

The environment design is an object oriented structure built on the C++ programming language. The design describes object representations that are used for the different classes of objects in the environment. Object representations are designed to provide a direct and useful interface to environment capabilities and programming constructs. The report also provides discussions of user interface, IU routines, inference, database and other capabilities, and how these facilities are integrated with the object representations.

The goal of the object oriented development approach is to build C++ objects to support interpretation of spatial and temporal data. This is primarily targeted for IU applications, though the techniques are applicable to other applications where reasoning about complex data is involved. C++ was chosen because it is a widely used, efficient, object-oriented extension of C, that facilitates the integration of public domain code, commercial programs and hardware devices.

The core set of C++ objects serves as a foundation for the representation of spatial, temporal and symbolic entities central to application development of IU and decision-aiding systems. It is intended that application developers will extend the object classes to create objects customized for their application.

The typical application IU system requires signal and/or image processing, symbolic reasoning, and inferencing capability, an interactive user interface for inspecting and manipulating the processing results, and an associated database for storing the original data and the derived results. Application developers require the following basic capabilities in their development environment:

- Objects that represent spatial/temporal data and models
- Objects that represent reasoning/inference knowledge
- Interactive display capability
- Remote storage and retrieval capability

The application developer can extend the baseline of objects and methods as required by their particular application. These objects promote the interoperability of higher-level C++ modules that conform to them. Figure 2 shows the relationships between these environment concepts.

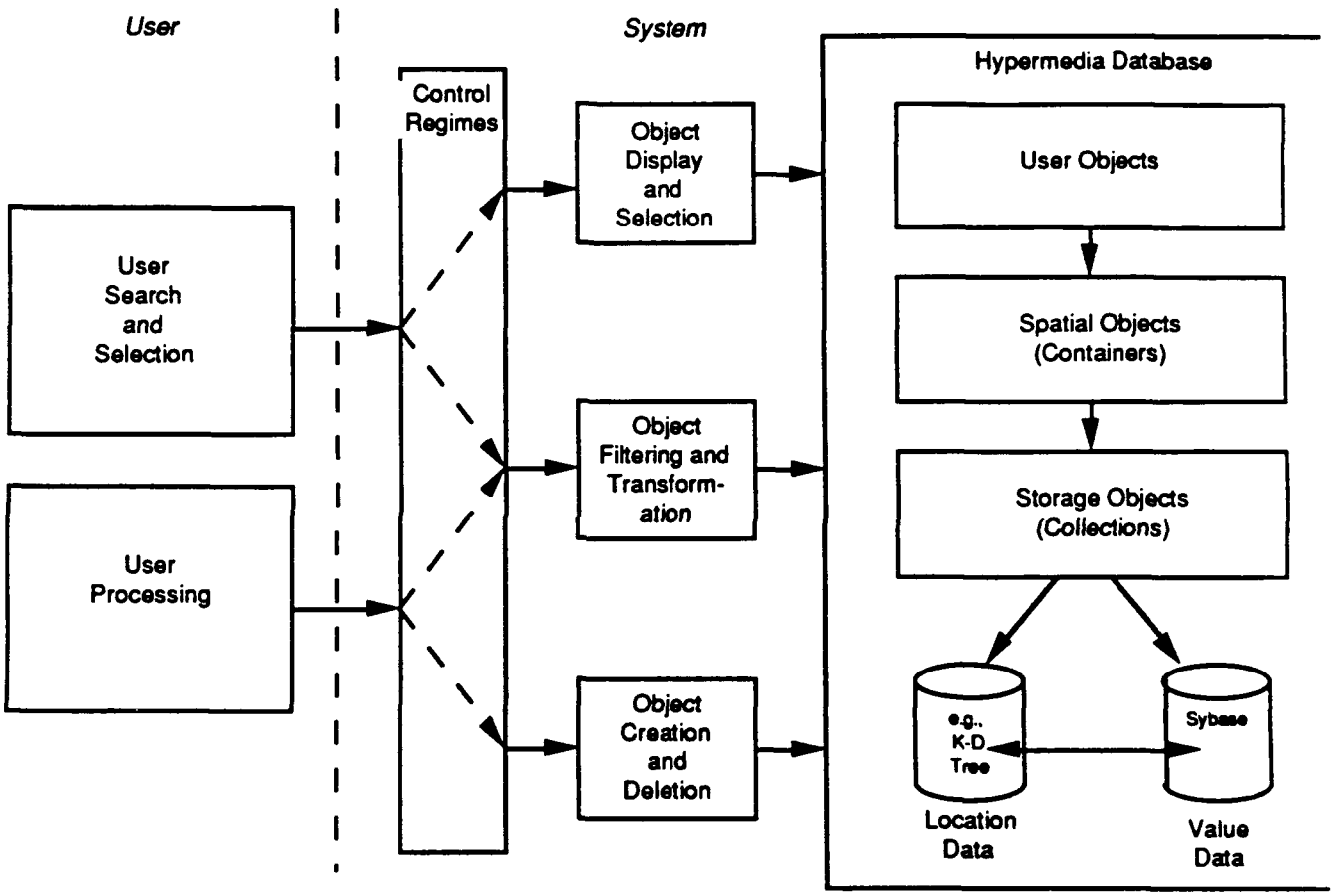


Figure 2: Vision Environment Architecture

The remainder of this document presents our design of these C++ objects and their associated class hierarchy. The hardware and software environment assumptions are described in Section 2. Section 3 describes the core spatio-temporal object classes, section 4 describes user interface objects and methods, and section 5 discusses database classes. Together, these objects comprise the major functional components of the design. Section 6 shows the top-level set of code libraries. Code libraries can include objects, but much of a code library contains procedural routines for doing various tasks. These are mostly intended to be wrapped as methods for core objects, although it is possible for the application programmer to use them in traditional programming paradigms, and in novel uses such as functions for concatenation through mapping function wrappers (section 3.4.1.2). Section 7 presents key

implementation issues that are considered in the design, including our approach to access of external databases, and the addition of code (sub-)libraries.

Terrain analysis and medical IU applications are presented in section 8. Terrain feature extraction from imagery and tissue segmentation in radiographs are analyzed to distill a common core of functionality required in the IU environment. This core provides the reusable infrastructure of the IU environment. Initial results are shown, including the re-implementation in the C++ environment of a Lisp/C program that performed model-based segmentation of hand radiographs using Bayesian inference for accrual of feature evidence.

Appendix A lists the core IU object class structure. Appendix B lists public domain image processing software packages that were considered for integration into the environment.

2.0 IU Environment Approach and Capabilities

Any IU environment aspires to support or possess all of the following goals and attributes.

- availability of algorithms
- execution efficiency
- interoperability
- verifiability
- portability

- extensibility
- coding efficiency
- function/data composability
- customizability

Efforts on other IU research and technology transfer environments [Quam, 84, KBVision, 87, Lawton and McConnell, 88, Lawton and Levitt, 89. Waltzman, 90] suggest that the first five goals are of primary importance for environments aimed at development of robust IU applications using well-understood IU technologies, i.e. technology transfer, while the second four are goals associated with rapid prototyping efforts common in IU research and innovative development of IU technology. This effort is focused at the goals that foster technology transfer.

In the following, we summarize hardware and software choices for a nearterm system build, then describe the fundamental philosophies and techniques for environmental software development. The third subsection presents results of programming instances designed to develop, test and demonstrate applications of the environmental constructs.

2.1 Workstation Hardware and Software Choices

Because of the bias towards technology transfer, and the desire to produce this environment within two years, environment component choices have largely been driven by current availability and prevalence of use of hardware and software options. Another driving factor was that as much as possible of the environment should be public domain, so that source code can be provided at minimal cost.

The basic development and user system is a Sun 3, 4 or Sparc workstation with a minimum of 12 megabytes core memory, keyboard and mouse, a color display and at least 300MB magnetic or read/write optical disk. A Vitek image processing acceleration board is under consideration for inclusion in the environment.

The software development environment is the Berkeley Unix 4.2 operating system on a Sun workstation, though compatibility to other Unix implementations and other workstations is maintained where reasonable. We have chosen C++ as the programming language. This choice is based largely on its efficiency, its relatively good compatiblity with C, and its nearterm widespread acceptance in the technology transfer community, i.e. the non-academic IU application development community. We have chosen the Free

Software Foundation's Gnu Compiler over AT&T's 2.0 C++ compiler for two reasons. The first is that the Gnu compiler generates faster, more efficient code because it is a true compiler and not just a preprocessor to a C compiler. The other reason is the availability of the compiler source code makes it portable to forseeable future (Unix) platforms.

X Windows is used for managing displays. The InterViews toolkit from Stanford provides a C++ interface to the X Windows package. IDraw, another Stanford product, provides the interactive graphic window interaction. Khorus, a public domain image processing library from the University of New Mexico, provides both the standard set of image processing functions as well as 2D plotting capabilities. Other public domain software packages being integrated in the basic environment include the CLIPS logic engine and rule-base package, the NCSA 3d display routines, and several neural net packages.

2.2 Core Spatial and Temporal Class Hierarchy

The class hierarchy is based on the structures developed in PowerVision and View [McConnell et. al., 88, Edelson et.al., 88]. In particular, the basic hierarchy of spatial classes and the concepts of transforms, function concatenation, virtual function wrappers, and programmable database-like search for perceptual grouping were all present in the original PowerVision implementation.

The current design has made strides in uniformity of these structures, cleaned up the relationship between objects and their display methods by associating display methods to the display objects (e.g. windows) rather than the source objects (e.g. a polygon), and has added class structures for coordinates. This design creates fundamental links between the geometric structure implied by coordinates and the programmability of search for perceptual grouping, as well as the linking together of lower dimensional spatial structures to form higher dimensional structures.

The core objects are organized into four general classes: scalars, collections, containers and coordinates. The scalars are the standard numerics and symbols of C++. Collections are general groupings of objects including arrays, streams, and graphs. Containers are groupings of objects that necessarily have an implied dimensionality and corresponding coordinate systems and imbedding spaces. Containers are inherently spatial: images, curves, solids, voxels, polygons, etc. Coordinates are objects that represent coordinate systems. Local coordinates are objects that are necessarily included within other objects (including other coordinates), while global coordinates can be disembodied.

Containers are designed to wrap around collections, and embed them in a coordinate system. Loosely speaking, we think of the semantic objects in IU systems, such as images, surfaces and volumes, as collections of values associated with coordinate systems. The grouping together in a systematic way of collections with coordinates forms containers. An array of integers is a collection. An array of integers associated with coordinates indicating the context of the array in pixels and centimeters is a container that is of the class Image. Figure 3 shows how containers, coordinates and collections relate to each other, and how they fit into an overall system.

Containers necessarily have coordinate objects and are closely tied to the user interface. The coordinate systems of the containers can map into the display coordinate systems. The display window itself is represented as a container. The necessary projections, translations, rotations, and scaling are

implemented by "virtual" containers that wrap around previously instantiated containers and convert them into the appropriately appearing object.

Collections do not have associated coordinate objects, although they can have indices, such as indexes for an array. Collections are closely tied to the underlying devices. For example, a collection can be made to correspond to a device such as an image scanner. The scanned image becomes an array (one representation of a collection). Efficient access, traversal and transformations are built as methods on collections. Another example is a neighborhood operation like convolution. It can be realized as a collection of data and a method that manages buffers to create fast virtual memory access to the data in the collection.

Tranforms are procedures that operate on containers, coordinates and collections and produce containers, coordinates and collections as output. Although it is possible to represent transforms as containers, providing a pleasing uniformity of data types, it can be semantically confusing to the user. Because technology transfer is a fundamental goal, we have erred on the side of clarity rather than uniformity. So an image is called an image, for example, instead of a function that represents a 2d surface in 3space. We intend to overload class names to permit users both views of appropriate objects.

Where it is not confusing, transforms are represented as overloaded constructors of the class of their output objects. For example, a histogram is a constructor method for the one-dimensional signal that is the output of the histogram transform on an image.

When possible, transforms are defined on containers but implemented on the (coordinate-free) collections to maximize reusability. For example, a one-dimensional smoothing filter can be implemented on an array, then be usable on any linear collection of data, such as an image row, a curve in 3 space, or a specific traversal of the edges of a solid. So the filter can be represented at the more abstract level of the container hierarchy as a method on a curveNd (i.e. a one-dimensional curve in N space), enabling polymorphism.

In section three, collection, coordinate and container objects are described in detail. There is also a description of how containers and collection objects are efficiently traversed, accessed and searched.

2.3 Current Environment Capabilities

The current environment status represents five months of design and three months of implementation on a 27 month effort. The Khorus image processing package has just become available as of the writing of this report and is not yet integrated in the environment Therefore, implementation results focus on basic user interface and database capabilities.

InterViews and IDraw are public domain object oriented user interface toolkits built on top of X Windows. To date, ADS has extended the graphical object hierarchy of InterViews and IDraw in two ways: the addition of images and of Bayes nets.

Within IDraw, images are first class objects. The user can put an image object into the drawing by clicking with the mouse and pulling out a rubber rectangle to define the outline of the image. The system presents the user with a menu of files, and when the image file has been chosen, inserts the image into the designated rectangle in the drawing, clipping the image if necessary.

Once an image object is defined and displayed, the user can perform a variety of drawing operations on it, as with any drawing object. Images can be moved, scaled, stretched in width or height, or rotated. Arbitrary image

warping is currently being implemented. In addition the user can draw any kind of graphical object on top of the image, and then group the object with the image, allowing drawing operations to be performed on both objects simultaneously. For example, the user can draw a colored polygon over a region of interest on an image, then group the polygon with the image into a composite object, then scale and rotate the composite object. The polygon still covers the same area of interest on the image. These capabilites are shown in figure 4. (To save space, some photos have been cropped so that the full computer screen is not shown.)

Rather than detailing each additional capability, we show our current state of implementation through two interactive processing scenarios. They demonstrate the benefits of an integrated object hierarchy, the use of images as first class objects, uniform representation of display and interactive object manipulation, and seamless access to remote processes.

The first application is diagnosis of arthritis from evidence extracted from a hand xray pictured in figure 5. Nodes and links are included as graphical objects. Graphically accessible methods are associated to form, in this example, a Bayes net object. Evidence can be acquired from images by measurement, and the evidence propagated through the Bayes net. Probabilities can be graphically inspected. For example, given a Bayes net that draws inferences about a disease condition of arthritic hands called periarticular demineralization, it is possible to take measurements on an xray of a hand in order to obtain evidence for the Bayesian network.

As illustrated in figure 5, the user loads the xray as an image object, draws a line down the middle of one of the finger bones (phalanges) and asks for a plot of the intensity values under the line by selecting "Profile" from a menu. The plot is shown in a window. A measure is taken of the relative density between the ends of the phalanx and the average density along the axis. This measure is added to the Bayes net as evidence by selecting the image, line and relevant Bayes node and selecting "Add Evidence" from the list of Bayes net menu options. The impact of the evidence at any point in the net can be seen by selecting the desired node and the menu choice "Show Belief". It is displayed as a probability histogram over the possible hypotheses at the node. In figure 5 these hypotheses are "demineralization" and "normal".

The second application scenario involves interactively querying a digital terrain database stored in Sybase. Figure 6 shows seamless interaction with an external process through a graphical interface. The user brings in an image of a map that is registered with the digital database. A region of interest is selected by drawing an ellipse on the map. Selecting the appropriate terrain layers and the "Retrieve" option from menus, a message is sent to Sybase, generating an SQL query. In this case, the database is populated with data on offshore oil wells, so a popup window of the wells in the region is displayed when the query results are returned.

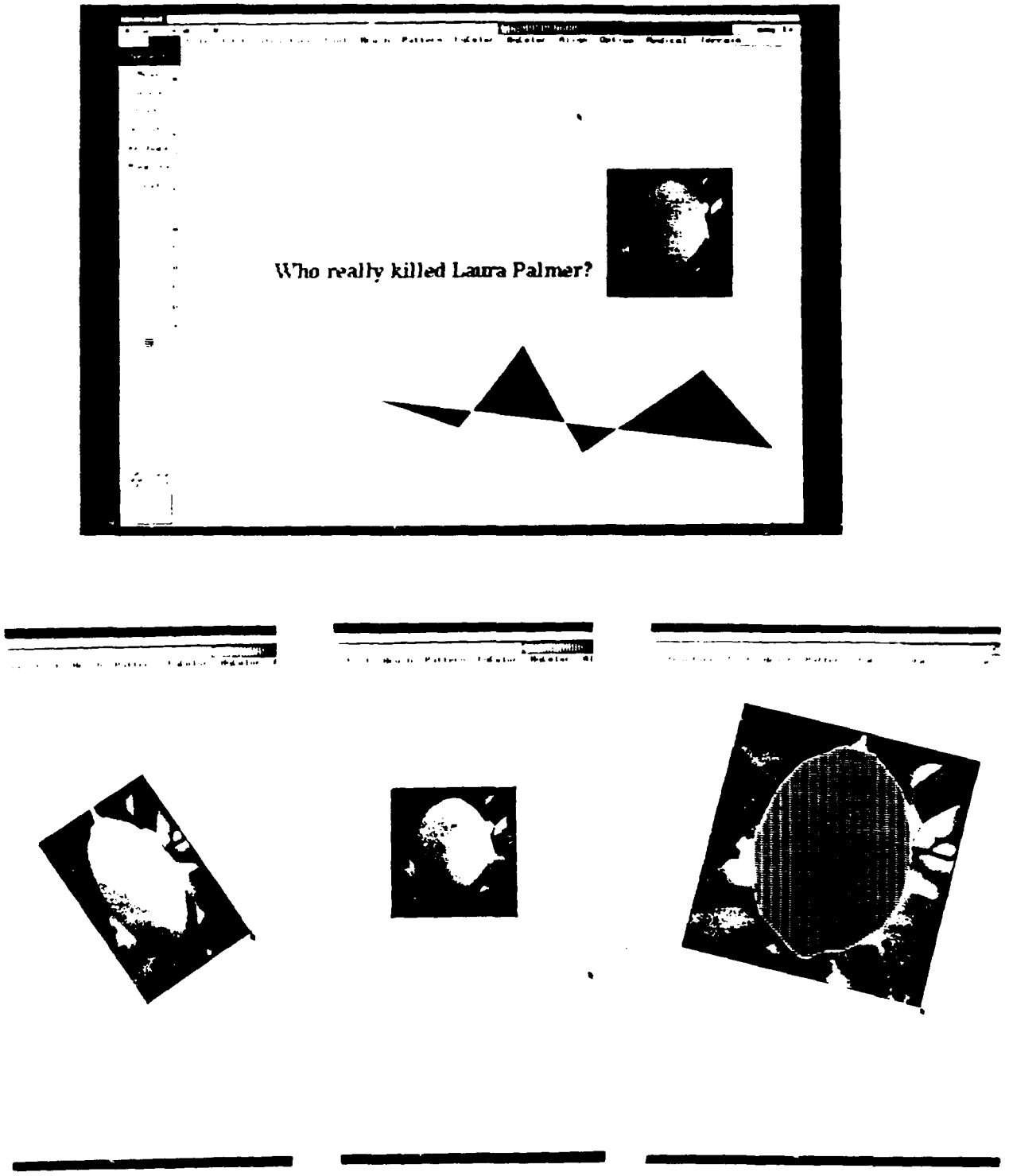
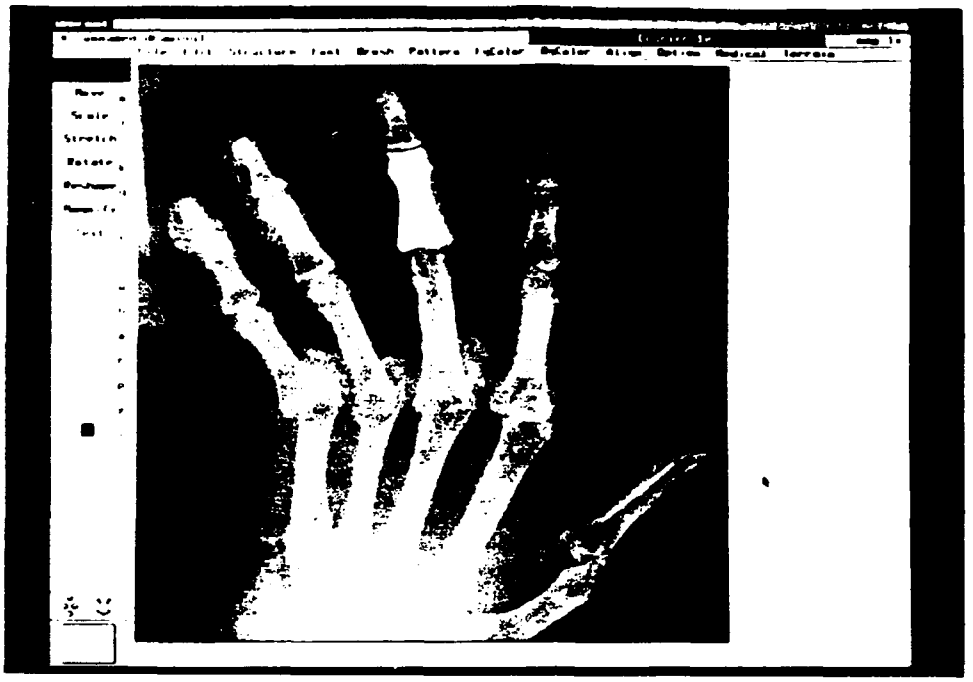


Figure 4. Image Display, Manipulation and Graphic Overlay



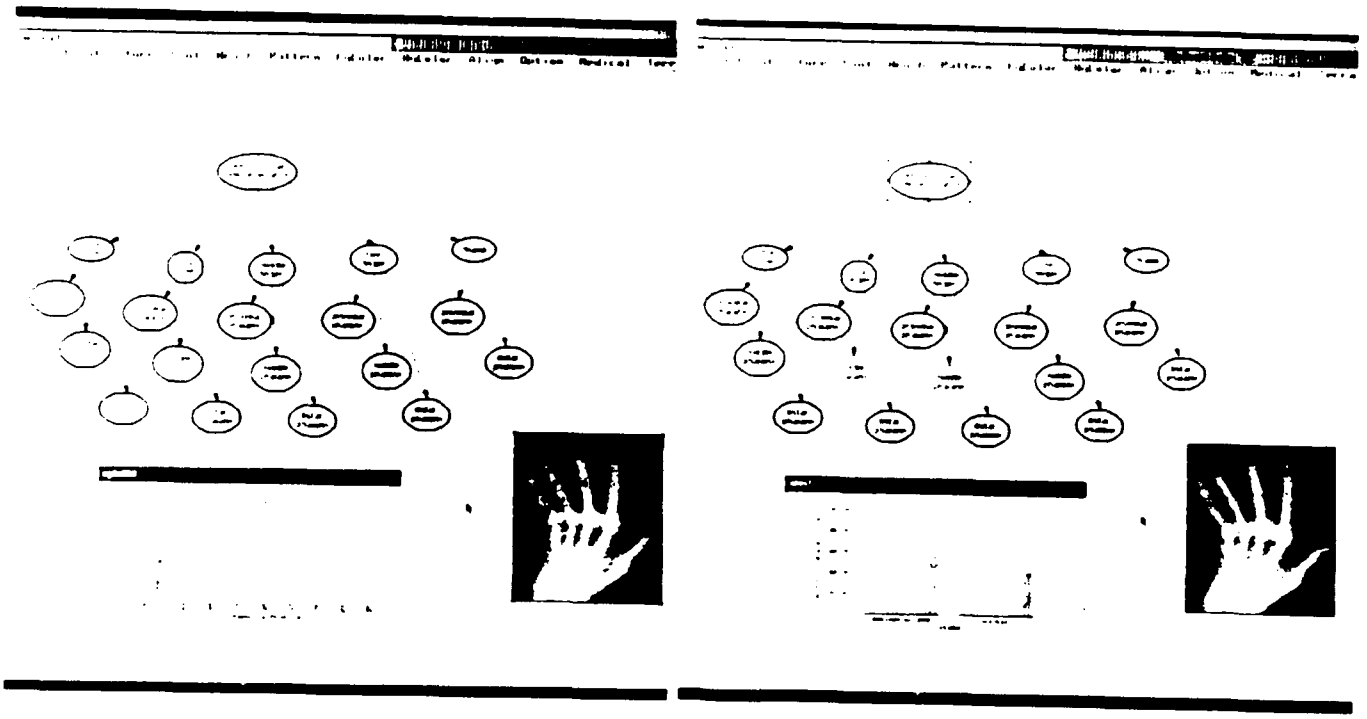


Figure 5. Integrated Image Feature Extraction and Bayesian Inference

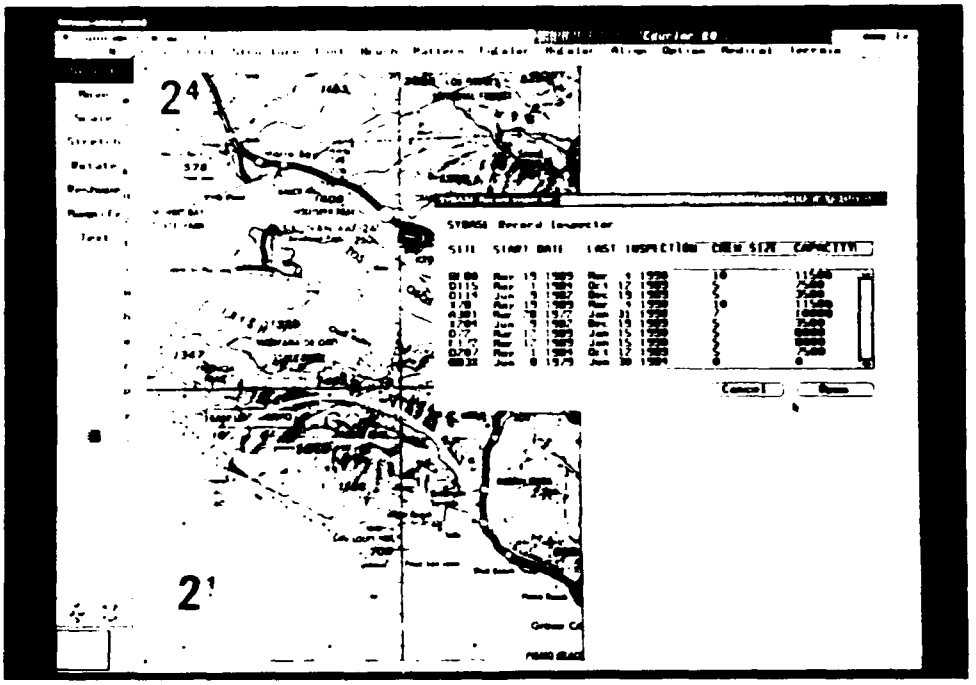


Figure 6. Graphical Sybase Interaction

3.0 Core Spatial and Temporal Objects

The core objects are high-level data structures in the technology domains of interest: image/signal interpretation, model-based reasoning, and hypermedia. The core objects are inherently hierarchical, with objects that decompose into member objects, that decompose into further objects, until the inner-most scalar objects, or values, are obtained. One of the advantages of the hierarchical representation is that it takes advantage of the (multiple) inheritance of attributes and methods that is built into C++.

The core objects are organized into four general classes: scalars, collections, containers and coordinates. The scalars are the standard numerics and symbols of C++. Collections are general groupings of objects: arrays, streams, graphs. Containers are groupings of objects that necessarily have an implied dimensionality and corresponding coordinate systems and imbedding spaces. Containers are inherently spatial: images, curves, solids, voxels, polygons, etc. Coordinates are objects that represent coordinate systems. Local coordinates are objects that are necessarily included within other objects (including other coordinates), while global coordinates can be disembodied.

Containers are designed to wrap around collections, and embed them in a coordinate system. Collections and coordinates are used in various ways to build containers. Figure 3 shows how containers, coordinates and collections relate to each other, and how they fit into an overall system.

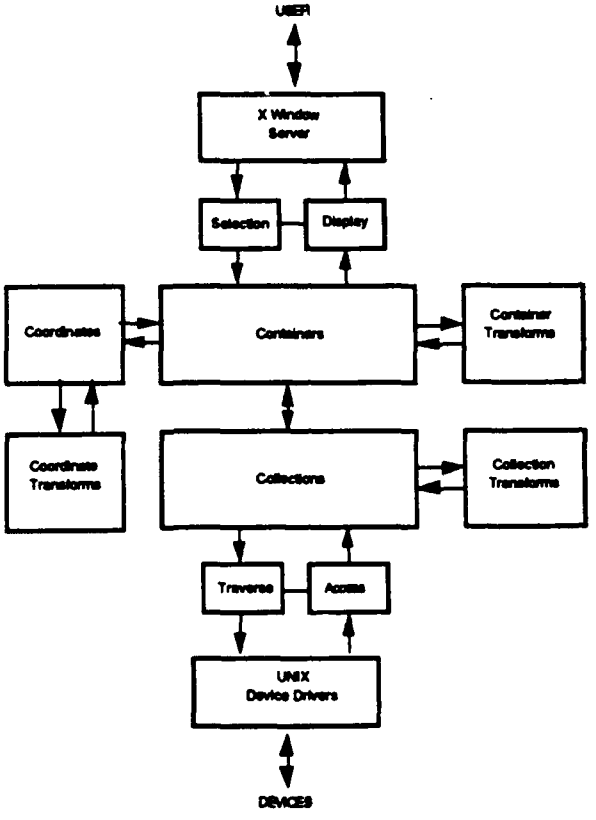


Figure 3: Core Object Relationships

Containers necessarily have coordinate objects and are closely tied to the user interface. The coordinate systems of the containers can map into the display coordinate systems. The display window itself can be represented as a container. The necessary projections, translations, rotations, and scaling are implemented by "virtual" containers that wrap around previously instantiated containers and convert them into the appropriately appearing object.

Collections do not have associated coordinate objects, although they can have indices, such as indexes for an array. Collections are closely tied to the underlying devices. For example, a collection can be made to correspond to a device such as an image scanner. The scanned image becomes an array (one representation of a collection). Efficient access, traversal and transformations are built as methods on collections. Another example is a neighborhood operation like convolution that can be realized as a collection of data and a method that manages buffers to create fast virtual memory access to the data in the collection.

Tranforms are procedures that operate on containers, coordinates and collections and produce containers, coordinates and collections as output. Where it is not confusing, transforms are represented as overloaded constructors of the class of their output objects. For example, a histogram is a cosntructor method for ValuedCurve1d that is the output of the histogram transform on an image. Where possible, transforms are defined on containers but implemented on the (coordinate-free) collections to maximize reusability. For example, a one-dimensional smoothing filter can be implemented on an array, then be usable on any linear collection of data, such as an image row, a curve in 3 space, or a specific traversal of the edges of a solid. So the filter can be represented at the more abstract level of the container hierarchy as a method on a curveNd (i.e. a 1 dimensional curve in N space), enabling polymorphism.

The next three subsections describe the collection, coordinate and container objects in detail. This is followed by a description of how containers and collection objects are efficiently traversed, accessed.and searched.

3.1 Collection Objects

Three classes of collection objects are planned: Stream, Graph, and Array. They can be characterized by the style of traversing and accessing the collection. Streams are traversed in a sequential manner, where the next access is restricted to the neighbor in a single forward direction. Graphs are traversed in a linked manner, where the next access is restricted to nearest neighbors in any direction. Arrays are traversed in a random manner, where the next access is unrestricted. IS-A hierarchy of collections in Figure 7.

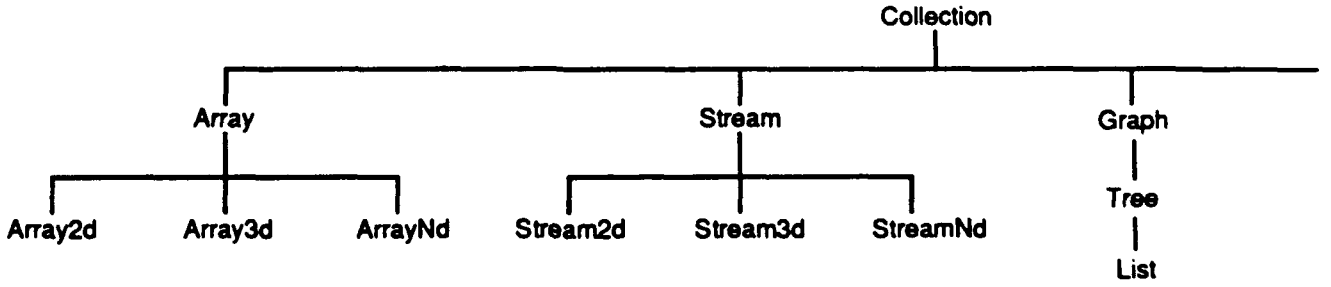


Figure 7: Collection Objects

3.1.1 Collection Classes

Any collection can group any set of core objects. Arrays can collect other arrays, or streams, or graphs, or scalar objects. Specific subclasses and/or methods are supplied for optimizing homogeneous sets of scalar objects.

The collection class hierarchy can be extended to wrap a stream, graph, or array around external sources. A character stream can be wrapped around a serial port. An array can be wrapped around a frame buffer. A graph can be wrapped around a Connection Machine or Transputer topology.

3.1.1.1 Streams

Streams are inherently linear collections. A series of objects is followed by an end-of-stream object. Higher dimensional streams are represented as nested streams. A buffered stream is supported to speed access of stream objects.

3.1.1.2 Graphs

Graphs are the general case of a linked data structure. A tree is a subclass of graph, and a list is a subclass of tree. This class hierarchy allows lists and trees to be manipulated by graph traversing routines, e.g.,"WalkDepthFirst". Graphs can store heterogeneous collections of objects. Graphs are implemented with separate node and arc objects. At this time we do not plan to support special subclasses for specific types

3.1.1.3 Arrays

Arrays are randomly-accessible object collections. The most general array is a linear collection of objects. Subclasses are defined that allow this linear collection of objects to be indexed with 2, 3, or N indices.

3.1.2 Collection Methods

The basic methods for a collection are as follows:

constructors	-creation and conversion routines
destructors	-memory/process/device deallocation routines
printers	-ASCII printing routines
traversers	-universal location generation routines
searchers	-selective location generation routines
accessors	-value access routines

"Constructors" allocate memory for a given object, then initialize this memory as required. This includes initializing the mechanisms for storage/retrieval of data to and from arbitrary sources (i.e. disk, display, digitizers, network, video disk).

"Destructors" deallocate memory, terminate, close and/or reset processes, devices and mechanisms as required.

"Printers" generate ASCII formatted representations of an object, both pretty-printed concise representations and full dumps.

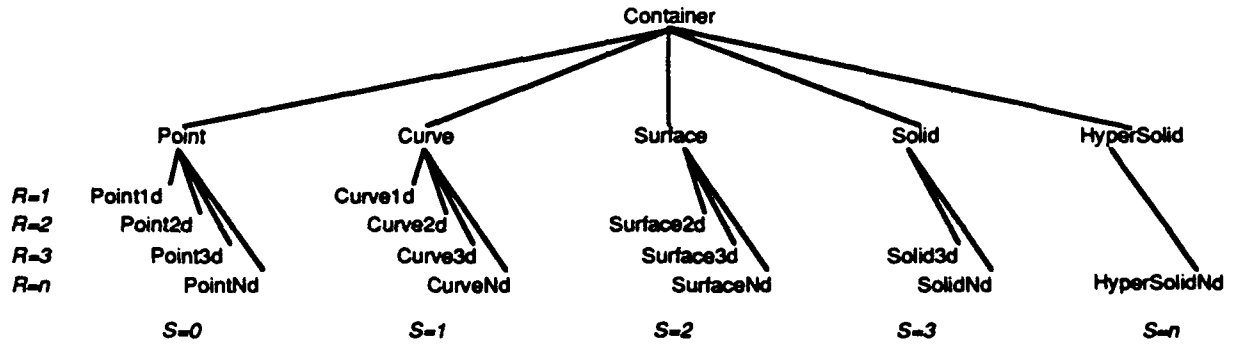
"Traversers" are methods for traversing the object, visiting each member object or element in turn. Each traversal of an object has an associated current location. Traversers accept a function pointer argument and apply the function at (a neighborhood of) each location.

"Searchers" are incremental, partial traversal methods. A search routine operates at the current location and chooses which location(s) to visit next., A search routine accepts two function pointers, applying one to the current location (neighborhood) and the other to choose the next location(s).

"Accessors" are methods for accessing the member object stored at the current location in the collection object. Access can be by value, or by reference to allow for overwriting.

3.2 Container Objects

Container objects represent an S-dimensional containment of objects in R-space.



This is only the top of the container class hierarchy. Additional subclasses are derived so that the leaf node classes of the hierarchy are realizations of more familiar spatio-temporal data structures and procedures. A signal is subclass of a one-dimensional container in 1-space. An image is a subclass of a two-dimensional container in 2-space. A volumetric representation is a three-dimensional container in 3-space. Constraints for a linear programming problem can be viewed as an N-dimensional container in M-space.

The container subclasses are effectively parameterized by the dimensionality of the container's topology, and the dimensions of the imbedding space. A 2d curve is a one-dimensional container in 2-space. A 3d curve is a one-dimensional container in 3-space. A polygonal region is a two-dimensional container in 2-space. A polygon3d is a two-dimensional container in 3-space. A terrain elevation map is a two-dimensional valued container in 2-space.

Specific classes of container are represented in multiple ways. For example, a three-dimensional container in 3-space is a solid, and a solid can be represented functionally (X^2 + Y^2 + Z^2 <= 1), volumetrically (via voxels or octtree), or by a surface model.

The cross-product of the S-dimensionality of the containers and the R-dimensionality of the embedding space is represented by a class hierarchy where the top-level branching is container dimensionality and the lower-level branching is embedded

space dimensionality. Point, Curve, Surface, Solid, and HyperSolid are the superclasses, and their subclasses correspond to the space the container is in.

To achieve efficiency, 0d, 1d, 2d, and 3d containers are implemented as special-cases, and Nd containers are handled in a general fashion. In the same manner, containers embedded in 1d, 2d, and 3d spaces are implemented as special cases, and embedding in Nd is handled in a general fashion. Beneath each branch of the container hierarchy are three subclasses that reflect increasingly general ways of representing a container:

1- Constant Containers
2- Valued Containers
3- Connected Containers
4- Aggregate Containers

Constant containers describe the shape of a container without representing its values , or "contents". The shape is defined to be its geometric representation in Nspace, without values necessarily being defined at locations of the shape. Because a shape is geometric, it usually has a boundary that we call its "shape boundary" to distinguish it from other uses of the term. For example, a solid cylinder in 3space has a solid cylinder as its shape, and a hollow cylinder as its boundary shape.

We can represent a force field acting on the solid cylinder by associating the appropriate local magnitude and direction of the force field with each point of the shape. This is an instance of a valued container. Valued containers have a shape description and a content mechanism, whereby values such as scalars or more complex objects can be associated or stored with each shape location.

Connected containers group other containers, relating them with a series of coordinate transforms or other relations such as adjacency or attachment, and merging them into a single connected entity. A CAD model of a single car built from surface facets is a connected container. A smoothing pyramid is a connected container, where image objects are related (connected) by the order of the smoothing and sub-sampling operations that created them.

Aggregate containers group a disjoint set of containers into a single entity. The set of CAD models of all cars manufactured at a particular plant is an aggregate container.

3.2.1 Container Classes

3.2.1.1 Point

Point has four subclasses, Point1d, Point2d, Point3d, and PointNd. Each represents a single location, without length, area, or volume, in the particular space of that dimension. For example, a Point2d is a point in 2 space.

3.2.1.2 Curve

Curve has four subclasses: Curve1d, Curve2d, Curve3d, and CurveNd. Curve1d is a standard, one dimensional signal.

"Walk_locations" is the general traversing mechanism for curves. It walks down every pixel (voxel) that lie on the curve's paths. "Walk_vertices" and "walk_edges"

are supported as well for curves that are represented as collections of vertices or edges.

3.2.1.3 Surface

Surface has three subclasses: Surface2d, Surface3d, and SurfaceNd. A one-dimensional surface is not necessary. Surface2d represents images as well as the 2d regions used in 2D modeling and 2D graphics.

The traversing mechanisms for surfaces (and solids) are grouped into two categories:

1- shape boundary traversal
2- shape traversal

Only the shape traversals are defined as methods on the surface object. The shape boundary traversals are available as methods of the boundary object (a curve) that surrounds the surface object. The methods concerned with traversing the shape of a surface are:

walk_locations	traverse internal points of surface
walk_vertices	traverse graph of surface regions where vertices are nodes
walk_edges	traverse graph of surface regions where edges are nodes
walk_faces	traverse graph of surface regions where faces are nodes
walk_segments	traverse series of straight-line segments that make up surface regions

Specialized traversals are supported based on a particular underlying implementation of a container. For example, a run-length encoded container has a method of traversing by run-lists.

3.2.1.4 Solid

Solid has one subclass, called Solid3d for conformity. Lower dimensional solids are not necessary. SolidNd describes shapes in higher dimensions that have only 3d volume, and no higher dimensional "mass". Solid3d supports volumetric models, surface models, and functional models of 3d shapes.

To traverse the boundary of a solid, the shape boundary container(s) is extracted, e.g., surface2d, and the traversing methods of that container are used. To traverse the shape of a solid, these methods are applied:

walk_locations	traverse internal points of solid
walk_vertices	traverse graph of solid regions where vertices (junctions) are nodes
walk_edges	traverse graph of solid regions where edges of solid regions are nodes
walk_faces	traverse graph of the faces of solid regions
walk_solids	traverse graph of solids regions
walk_segments	traverse series of straight-line segments that make up solid regions

3.2.1.5 HyperSolid

HyperSolid has one subclass, HyperSolidND. Lower dimensional hypersolids are unnecessary. These can be represented with generalized volumetric techniques, surface representation (collection of hyperplane constraints), and functional representations.

Traverses are the same as for solids, generalized for N-dimensions.

3.2.2 Container Methods

The basic set of methods for a container are the following:

constructors	-creation and conversion routines
destructors	-memory/process/device deallocation routines
printers	-ASCII printing routines
traversers	-universal location generation routines
searchers	-selective location generation routines
accessors	-value access routines
draw	-draw representation of self in an Xwindow
display	-add self to display list (see section 4.2)
inside	-predicate to determine if point is inside boundary

"Constructors" are a primary mechanism for deriving new objects from old objects. Constructors can be used to create aggregate objects by grouping other objects. Constructors can wrap up other objects, and add functionality to transform them into the new object. A particular class can have several different constructors, each identified by its unique argument list. The arguments of a constructor can be by reference or by value. When objects are passed by reference to a constructor, the newly created object incorporates the old objects into itself. Its functionality then depends on the state of its internal objects (or the objects it references).

"Destructors" deallocate memory associated with an object, and in turn invoke the "destructors" of their constituent objects.

"Printers" generate various formatted ASCII output of the container. This is similar to the collection "printers" with the addition of coordinate information.

"Traverser", "Searcher" and "Accessor" methods typically window through to an underlying collection. The current position of a container traversal is in effect a current position of the underlying collection traversal, and the mechanism for accessing the data in the container is the same as the mechanism for accessing data in the underlying container or collection.

"Display", "Draw" and "Inside" are methods for realizing the user interface. See section 4 for details. The display method queues the object for display in an Xwindow by placing the object on the display list (see section 4.1) of the window. Then the window object takes care of determining the necessary parameters to call the object's draw method (see section 4.2). An object's draw method produces pixel values that are a representation of itself and maps them into a window display, or any container of type Surface2d. The inside method is used by the window object for each object on the display list to determine if a particular mouse click has fallen within its bounds (see section 4.4). Only containers and coordinates can be displayed in the 2D

and 3D object windows, as coordinates are required to relate to the window display. It is possible to display collections in structured text or graph browsing windows

3.2.3 Container Subclasses

The set of methods of each container class is the interface between objects of that class and the external environment. In a formal sense, the specification of these methods constitutes a contract with the external environment. The strong type definitions of C++ help ensure the correct form of all interactions with the object. When an object is passed as an argument to an arbitrary function, the set of methods define how the function can manipulate the object. When an older object is supplied as an argument to a newer object's constructor, the object methods operating on the older object present methods to the newer object that can then be exploited to realize the newer objects capability.

If the older object is passed to the newer object by reference, the two objects become related, and modifications to the older object are reflected in the functionality of the newer object. If the older object is passed by value (or the newer object makes an internal copy of the older object), then the two objects remain unrelated.

In general, the construction of containers involves the wrapping of a new container around a set of older containers. The class of the new container defines what it is. The classes recognized by the constructors define how it can be built. The information embedded in the container is a mapping from the containers supplied as arguments describing the functionality of the resultant new object.

The subclasses of the containers were designed with this in mind The primary description of any subclass is twofold:

1- description of the capability of the container it implements
2- description of the set of containers (or collections) it can accept to construct this capability

The resultant subclasses are prefixed with Constant, Valued, Connected and Aggregate. Each subclass describes a basic approach to building containers. Constant builds the shape of a Constant container. Valued borrows the shape of an existing container and inserts new values.

Connected takes a set of containers and relations between them and groups them into a single container. The set of relations must be "path-connected" in the sense that given any two of the containers, A and B, there is a sequence of containers in the set starting with A and ending with B, such that there is a relation specified between any two containers that are adjacent in the sequence. For example, a CSG model of a tank that had coordinate transforms and attachments specified between adjacent primitive solids is a connected container.

Table 1 summarizes the possible ways of constructing these various containers from other containers and collections.

Table 1: Constructor Arguments

Container Subclass	Constructor Arguments
Constant Container	1) another container that represents the shape
	2) an existing collection and corresponding coordinate information
Valued Container	1) another container for the shape and an existing collection for the values
	2) another container for the shape and an existing container for the values
Connected Container	1) list of containers and relationships
Aggregate Container	1) list of containers

3.2.3.1 Constant Containers

Constant containers are descriptions of a region but do not describe the values associated with points contained in the region. Typically, constant containers are implemented as a representation of the shape of the container. These containers represent a locus in some space. The boundary of the container can be traversed, the insides of the container can be traversed, but no values can be retrieved. Constant containers can be constructed from a lower-dimensional container that describes its boundaries. Two-dimensional surfaces are bounded by a closed two-dimensional curve. Three-dimensional solids are bounded by a three-dimensional surface (which is in turn bounded by a three-dimensional curve).

3.2.3.2 Valued Containers

Valued containers describe the region as well as provide a method for accessing the values associated the region. The boundaries and insides of these containers can be traversed, as with Constant containers, and values can be extracted at each location in the traversal.

In general, valued, or "full", containers can be constructed by combining a Constant container object with a collection that maps locations in the container to values. This collection (an array, stream, or graph) can represent values inside the new container, or be restricted to locations on the boundary.

This general mechanism can be overridden by specific full containers that rely on a technique that entangles the boundary representation with the value representation. In this case their constructors do not have other containers (or collections) passed to them by reference.

3.2.3.3 Connected Containers

Connected containers group other containers into a single entity. The generalized composing mechanism relates objects by chaining them with a series of relationships, such as coordinate transforms, in the appropriate space. 3d objects can be chained with 3d rotations, scalings, and translations. 2d objects can be chained with 2d rotations, scalings, and translations. Other grouping methods, such as symbolic groupings of INSIDE-OF and ADJACENT are realized with subclasses of the general composition class.

More explicitly, the set of relations passed to the connected container constructor must be "path-connected" in the sense that given any two of the containers, A and B, there is a sequence of containers in the set starting with A and ending with B, such that there is a relation specified between any two containers that are adjacent in the sequence. For example, a CSG model of a tank that had coordinate transforms and attachments specified between adjacent primitive solids is a connected container.

The general composition mechanism can also be used to transform the local coordinate systems of an existing container, by grouping it with a constant container associated with a different coordinate system. Projecting a container into a coordinate system with less dimensions is not handled by the connected container mechanism, but is instead supported by the constructors of objects in those lesser *dimensions*.

Constructors for connected containers take a list of older (sub) containers and associated local-coordinate systems (implemented as coordinate objects, see section 3.3), and other relationships, such as attachment and adjacency, as by-reference arguments. This list can include simpler containers in the same space, i.e. a connected two-dimensional surface can accept two-dimensional curves, because they are degenerate cases of two-dimensional surfaces.

3.3 Coordinate Objects

Coordinate objects represent coordinate systems. A coordinate has a corresponding type that is one of cartesian, polar, cylindrical, spherical, quaternionic , or shape. Shape means the coordinate system is defined in terms of distinguished points in a container, like attachment points, or the ends of axes of sub-objects. A local coordinate is necessarily contained in another object such as a container or another coordinate. A disembodied coordinate is defined to be the subclass of global coordinate. Coordinates have methods that act as transformations between other coordinate systems. A coordinate records its transformations between other coordinates, unless these transformations are explicitly deallocated.

3.3.1 Coordinate Classes

There are two subclasses: global and local. Local has a special subclass called base-coordinate.

3.3.1.1 Global Coordinate

Global coordinates can occur disembodied, i.e. without being contained in or referencing other objects. They can be transformed and copied by any coordinate

constructor to mix in when constructing a local-coordinate defined for a container or other global or local coordinate. This "places" the container in the global coordinate system. The global coordinate remembers the containers that were constructed with it.

3.3.1.2 Local Coordinate

A local-coordinate can represent the imbedding space of the container, or other ego-centered coordinate systems. An object can have multiple local coordinates. Each local coordinate that is constructed must have a transformation that represents it in the coordinate system defined by the base coordinate.

3.3.1.2.1 Base Coordinate

The base coordinate is a distinguished local coordinate. It is defined to be the first local-coordinate associated to a container or other coordinate. The base coordinate is instantiated by the container contructor. It can be specified by the caller of the contructor method. The base coordinate is guaranteed to have transforms associated to all other local-coordinates of that container or coordinate. It is intended, although not required, that the base-coordinate correspond to the natural traversal of an associated collection of values. For example, a raster image is a Surface2d container whose values are in an Array2d (collection). Its natural base coordinate is the cartesian coordinate with origin at array index (0,0), one axis in the row direction and another corresponding to columns. For a pyramid, a natural base coordinate is similar, but includes a third axis in the multi-resolution direction.

3.3.2 Coordinate Methods

Coordinates all contain the following methods. Note that when local and global are not explicitly called out, either applies. For example, the transform method can relate locals to locals, locals to globals or globals to globals.

type	- returns a mathematical type (e.g. cartesian) or the type "shape".
origin	- returns a point
dimensions	- returns list of dimensions
units	- returns list of named units per dimension
minextents	- returns list of minimum extents per dimension
maxextents	- returns list of maximum extents per dimension
convert	- inputs a type that is not "shape" and creates versions of its local-coordinates expressed in that type (e.g. cartesian to polar conversion)
list-transforms	- returns the list of transforms known between itself and other coordinates
transform	- inputs another coordinate with a known transform to itself, and a third coordinate with a known transform between it and the second coordinate; returns a transform between itself and the third coordinate.
propagate-transform	- inputs a coordinate with known transform between itself and the coordinate, and returns the list of transforms between all its local-coordinates and the input coordinate.

3.4 Object Traversal and Search

From the user's point of view, containers get traversed or searched. From the workstation environment's point of view, containers are pointers to collections that get traversed or searched. Both traversal and search can be thought of as routines consisting of the cyclic applications of three functions: move, access.and apply. In the case of traversal every value in the underlying collections is necessarily visited, so the function for moving, or choosing the next location(s) in a container's shape, is known before traversal is invoked.

In search all locations/values are not necessarily visited. The move function must be passed by reference to the search method. The apply function is invoked on the appropriate neighborhood at each visited location for both traversal and search methods.

Signal and image processing functions traverse their contents to enable extraction of higher-level interpretations. These routines need to quickly iterate across their N-dimensional data sources, with efficient access to a local neighborhood ranging in size from 1 to M units in any dimension.

Traversal is intended to provide the support for a programming style whereby the application developer codes the operation to be done at each point in the traversal, and leave it up to some other mechanism to slide this operation around the container. This requires two things: an underlying mechanism for efficient traversing (tied to an efficient accessing scheme) and a programmer interface.

Examples of underlying mechanisms are the tiling of imagery used in ADRIES, and the sliding-window subsystem from the Honeywell Image Research Laboratory. Each makes neighborhoods of pixels available to the programmer in an efficient manner.

3.4.1 Programming Traversal and Search

When a programmer is presented with an efficient source of neighborhood data, it is convenient to string together a series of smaller functions to do the work of a more complex function. However, the programmer is typically forced to write the complex function out flat, inline in one function, to avoid the overhead of piping data between functions. The IU workstation environment provides support to concatenate existing low-level operations without incurring extra overhead.

This can be done by constructing a library of neighborhood operations that describe what is done on one neighborhood, but contain no mechanism for iteration. Examples are convolution kernels, median filtering, and basic arithmetic and logical manipulations of Nd data. Neighborhood operations that maintain a state are implemented in this model by saving the permanent state in static and/or global variables for later retrieval.

This makes the process of writing more complex neighborhood operations into one of concatenating the series of operations into a single neighborhood operation. A specific neighborhood function is then inserted in the middle of a looping mechanism that is capable of traversing the container, and supplying the neighborhoods of data to the operator.

There is a split here that needs to remain clear. The reusable neighborhood operations are small code chunks that have no built-in looping mechanisms. An application specific neighborhood operation takes a sequence of reusable neighborhood operations, and wraps them up with a traversal or search mechanism. To reuse that specific neighborhood operation means the body of the loop has to be extracted and made into a stand-alone function. That effort is only desirable when the function is to be reused; otherwise it leads to extra overhead from a layer of function call. An example of convolution code is shown below. Convolution has a neighborhood application function, and an outer loop that traverses a container.

```
int ByteStream2d::convolve
(
   ByteStream2d *outstream;  // Output 2d stream of bytes
   ByteArray2d *mask;        // Convolution mask.
)

{

// Local variables
   int mask_width = mask->width();
   int mask_height = mask->height();
   int dummy;

// Ensure input and output streams are rewound
   this->rewind();
   outstream->rewind();

// Create and fill neighborhood cache on input stream
// Boundary handling is done by the cache.
   char **instream_cache =
      this->cache( mask_width, mask_height );

/* Load entire mask into its own cache */
   float **mask_cache =
      mask->cache( mask_width, mask_height );

// Loop until double end-of-stream
   while( !instream->eos() )
   {

   // Loop until single end-of-stream (end of row)
      while( !this->eos() )
      {
         int out_value = 0;

      // Convolve mask with neighborhood
      // and write result to output stream
         for( int i=0; i<mask_height; i++ )
            for( int j=0; j<mask_width; j++ )
               out_value += *(*(mask_cache+j)+i) *
                            *(*(instream_cache+j)+i);
         outstream->next() = out_value;
         dummy = this->next();
      }

   // Set up for next row
      dummy = this->next();
   }

// Return OK status
   return 0;
}
```

3.4.1.1 Neighborhood Cacheing

Object access is streamlined with a cacheing mechanism that makes a local neighborhood available to the C++ program in an internal C++ data structure. The mechanism is program-controlled, in that the program decides when to initialize it and when to refresh its contents. Because the cache is represented as a standard C++ data structure, either an array or a nested array of pointers to arrays, the efficiency of data access within the cache is identical to array-based data access.

A neighborhood cache is useful for representing windowing operations on imagery. The input object is an image, the cache is an array of pointers to linear arrays; as the window slides across the image, the cache is refreshed by updating the pointers.

For point transformations the cacheing mechanism is useful in order to reduce the overhead of row access. The cache is defined to be a single array equal in length to the image row, and it is refreshed after each row is processed. The processing of the row is done with a tight for-loop, with entirely in-memory data access.

The convolution example above illustrates the use of neighborhood cacheing for arbitrary convolution of imagery. Two caches are employed, one on the input image stream, another on the 2D-array that defines the convolution mask.

3.4.1.2 Mapping Function Wrapper

The ability to compose functions without creating intermediate data structures yields the ability to display the results of experiments with a minimum of typing on the part of the programmer (e.g. not creating named functions as above) and with a great saving of memory and memory management overhead. In Powervision (the ADS vision development environment built in ZetaLisp on a Symbolics lisp machine), functions created this way were called "pixel-mapping-functions". Because individual objects know how to display themselves, the pixel-mapping-function capability is re-created with a wrapper that says: compose the following functions (passed by reference) on this input data, and add the display method for the result of the last function at the end. Ordinarily, the final result is saved, because it is often the input to a next stage of processing.

For example, to apply a convolution to only the pixels in an image defined by a mask, a search method is applied to the image object, where the search method run length encodes the mask and accesses the "on" pixels only. The search method is composed with the convolution to feed only the relevant neighborhoods to the convolution kernel.

Mapping functions are implemented by subclasses of their respective containers. There are four basic types of mapping functions, and so four basic mapping function class extensions:

1) coordinate transformation of container locations
2) look-up-table applied to container values
3) arbitrary expression applied to container values
4) arbitrary expression applied to container locations.

3.4.2 Spatial Data File System

Two capabilities have been defined to support the traversal and access of spatial data stored in containers: 1) an efficient file access mechanism, for retrieving data stored on arbitrary devices, and 2) efficient indexing schemes for quickly locating data stored within a container. In combination these are called the Spatial Data File System.

Collection objects are supported on diverse sources of data: disks, frame buffers, digitizers, optical disks, even virtual memory. Constructors set up and initialize access to these devices, and subsequent use of the object's methods so that the underlying access mechanism is transperant to the application developer. Destructors close and/or reset device access as needed.

Device drivers simplify this transparency, by making all objects appear as a stream of characters (or a buffered stream of characters). Furthermore, the model of a disk device driver suffices for a large subset of devices that can be viewed as a contiguous collection of characters coupled with a random seek mechanism.

The spatial data file system supports the I/O of spatial data to disk-like devices. The spatial data file system differs from a normal Unix file-system in that it attempts to optimize the access of large collections of 2d, 3d, or Nd data.

The ability to efficiently index into specific data stored within a larger collection is required. Database indexing uses (multidimensional) trees (B-tree, kd-tree, quadtree, octree, etc.) or hashing to isolate a particular item in a list.

For example, a collection of curves written to disk can be implemented in the following manner:

1- an array of bytes on disk is defined as the low level object

2- a tree index is computed that maps from the curve i.d. to the byte offset within the file

3- an array of curves is defined that combines the array of bytes with the curve-to-byte tree, and results in an array of efficiently accessible variable length curves that (just happen to) reside on disk

4.0 User Interface

The user-interface supports the direct manipulation and inspection of all entities in the vision environment through a windows-menu-and-mouse interface. The user interface is based on X Windows, a network window system, and InterViews, a C++ package that defines basic X Windows objects. The user interface must be capable of displaying a list of containers to an X window, and mapping mouse clicks to specific objects in the display list. The following subsections present the display list object, window types and addresses issues in imagery display and mouse protocol.

4.1 The Display List

The display list is an object that keeps track of what set of objects is currently being displayed in a window. There is a single display list associated with each window. The display list is implemented as a connected container of 2d surfaces. The connected container groups two-dimensional points, curves, and surfaces, interrelating them with coordinate transforms and other programmed relations. Each container stored in the display list knows how to redisplay itself, and knows how to determine if a given point is inside or outside its 2d shape boundary or whether a given rectangle overlaps its shape boundary.

4.2 Windows

The window system supports overlapping windows, as well as neatly tiled windows, useful for applications once they reach a certain level of maturity.
While in overlapping mode, each window can be resized, repositioned on the screen, collapsed down to an icon, and expanded back to its original size and shape. When overlapping mode is disabled (tiling mode), the size and shape of each window is predetermined (or tightly controlled), and the opening and closing of windows is directed by the application. Each window is associated with a display type that governs the type of information that can be shown within the display region.

The supported window types include:

1. **2D Object Display Window**-- images, lines, curves, 2D graphics
2. **3D Object Display Window**-- volumes, surfaces, 3D graphics
3. **2D Plotting Window** -- 2 axis: signal plotting, time series analysis, statistics, measurement or feature spaces in two dimensions
4. **3D Plotting Window**-- 3 axis: measurement or feature spaces in three dimensions
5. **Directed Graph Browser Window**
6. **Structured Text Browser Window**
7. **Dialog Box Window**

The display of objects to windows is object-oriented, in that each entity within the vision environment knows how to display (or present) itself to a window of a specific type. The following subsections discuss each window type.

4.2.1 2D and 3D Object Display Windows

An object display window presents a collection of image-understanding objects to the user, all registered to a single coordinate system with the display based on a single viewing perspective in that coordinate system.

The coordinate system of a 2D object display is that of the base coordinate of the primary image associated with the display. The typical viewing perspective of a 2D object is such that the primary image is displayed at full resolution. If the window size exceeds the primary image size, the remainder is filled.with a constant value If the primary image size exceeds the window size, the image is clipped. From this starting point, the viewing perspective can be adjusted to arbitrarily zoom and scroll the display.

3D display involves projecting 3d objects onto a 2d space, then entering the resultant container into the display list. This projection is done by the constructor routines of the 2d classes. For example, a constructor for 2d curves is supported that accepts a 3d curve and a set of coordinate transforms that describe the relationship of the 3d object's coordinates to the screen of the desired 2d projection. The resultant 2d container can transform surface orientation into intensity values of 2d surfaces. Hidden surfaces are dealt with by this projection mechanism as well.

The coordinate system of a 3D object is centered around the origin of some primary object associated with the display. The viewing perspective is based on the unit screen at a unit distance from the viewers focal point. A default 3D viewing perspective can be defined as a global coordinate object and placed at a distance from the viewer where the bounding rectangle of the unit screen matches the bounding rectangle of the object (padded with constant values to maintain aspect ratio) projected onto it (i.e, it fills the screen). From this starting point, the unit screen can be translated and rotated in 3-space to a new position that results in a new projection of the 3D object onto the unit screen, and a new display. Fant's warp and perspective algorithm is used for this method.

Each object in a 2D display is z-buffered to achieve an ordering from front to back, allowing for the overlay of graphical objects on top of other graphical objects (or images). This ordering also resolves which object responds when the cursor is positioned on it and the mouse is clicked. Ordering in 3-space is dependent on the viewing perspective, but for essentially 2D objects that lie on a 3D surface there can be a concept of ordering with respect to a particular side of that surface.

Any object that can display itself with the proper dimensions can be added to the list of objects displayed by a window. The following classes are supported for 2D: grey-scale images, binary images, labeled images, lines, curves, polygons, and any fixed projection of a 3D object to 2D space. Both volumetric and surface models are supported for 3D object display. Hidden line removal and other surface rendering techniques are supported as needed.

Color is required for drawing secondary objects on the face of primary objects, such as lines and curves embedded in an image, or the vertices of a 3D object that lie on its surface.

There is a need for "snap-to" capability that associates the position of the cursor when a mouse-button is clicked to the nearest reasonable object. This aids the user in selecting objects of single pixel width.

4.2.2 Plotting Windows

Plotting windows share traits with object display windows. The main difference is that they graphically present information that is inherently numeric, whereas object displays graphically present information that is inherently pictoral. The extent of each numeric dimension is presented to the user as an annotated axis, divided by tic marks into numeric ranges. By, default the plot is scaled to fit within the current window size.

2D plots consist of line plots, scatter plots (by dot and by character), and bar plots. Grids to aid in viewing are optional. 3D plots consist of surface grid plots (with hidden lines removed) or chunky bar plots.

In a similar fashion to object displays, any object that can present itself to the plotting window as a collection of numbers of the proper dimension can be added to the list of objects plotted by the window. The default ordering is FIFO, but can be modified by the user.

4.2.3 Directed Graph Browser Window

Arbitrary directed graphs, trees, and lists can be used within the environment to group together objects. The directed graph browser is provided as a general tool for graphically inspecting these data structures, and editing their contents. In general, the directed graph browser supports an arbitrary directed graph, but works just as well on the simpler data structures of trees and lists. It automatically positions the nodes within the window, drawing arrowed lines to show the relationship with other nodes, and asks each node to display itself, either by icon or by name.

Navigation of the graph can be done by the user or under program control. An advanced browsing feature is a miniature map of the entire graph, used to navigate when the graph is too large to fit in the window.

Nodes in the graph can be selected by clicking left. A menu of possible node operations is brought up by clicking middle, once the node has been selected. Certain node operations require a target node that is then selected by the right button after the first two operations.

The directed graph browser can be set up for read-only access of the directed graph, or with read-write permission can be used for interactive editing of the structure. Nodes can be created, deleted, and moved. In a similar fashion entire sub-graphs can be created, deleted, and moved. The structure of created sub-graphs is based on a list of default structures, or controlled by the program. Hierarchical graphs (nested) may be supported as well.

4.2.4 Structured Text Browser and Dialog Box Windows

The input/output of text and numbers is done through structured text browsers. At its simplest (initial capability), this is a text editor. At its most complex (eventual capability), it is a text I/O window that automatically enforces a certain grammar,

such as the legal relationships that can be entered into a semantic network, the syntax of a programming language, or a database table structure.

Once again, as in the directed graph browser, text display in this window can be either read-only or read-write. A read-only text display, augmented by certain control buttons, is a dialog box.

Actions can be associated with the modification and/or selection of any text within the window. By clicking on a word in a list of words, it is possible to bring up another window with information pertaining to that item. This rudimentary hypertext is useful for following a chain of related objects in the vision environment.

The user interface is an easy to use hyper-text or hyper-media interface, built around the data constructs necessary for IU: 1, 2, and 3D signals and structures, and diverse networks and databases to contain them. To support IU further, links between various displayed entities are arbitrarily complex transformations, instead of simple pointers, to make for a flexible approach to prototyping applications. A single plane in a 3D plot can be selected, and displayed in a 2D plot window. If the new window is created by reference (versus by value, to borrow programming language terms), a modification in the original 3D plot is passed on to the 2D plot.

4.3 Overlays and Sprite Objects

Most graphics that overlay images occupy a small area compared to the image size. An example is the overlay of linear features, such as roads or rivers, on an image. When the image is drawn, it is from one container object. Each window is associated with a display type that governs the type of information that can be shown within the display region. The linear features are assumed to be a second container object with coordinates that overlap the image. The problem is to allow the user to select and unselect the display of the graphic overlays without redrawing the whole screen just to refresh the small area under the graphic overlays. The approach is to create a third object that has the same container (i.e. "shape") information as the graphics, but uses the values from the image collection. Refreshing the screen is accomplished by requiring the appropriate set of graphic objects to refresh themselves. This capability is called a "sprite" object in the object-oriented imagery display literature.

4.4 Visual Pointer

The initial pointer device is a three button mouse, because of the choice of Sun as the initial hardware platform. A mouse is used to manipulate a cursor on the screen. Action is taken when a particular mouse button is pressed. The action taken is a function of which button is pressed, what window the cursor resides in (the top-most window if there is overlap), and where in the window it resides. Standards are being incrementally developed to govern the types of actions associated with each mouse button. The cursor may change in size or shape to indicate change of state of any particular application.

Each window comprises several mouse-sensitive areas within it. These areas may or may not correspond to obvious graphical clues, such as a menu item, button, or graphical object. For the most part the mouse-sensitive areas lie within the bounding rectangle of the window, with the single exception of pop-up menus (or detachable menus) that are attached to the borders of the window.

Mouse selection is implemented by capturing a mouse click, and interrogating each object in the display list in turn, to determine if the click fell upon it. The effective area of the click is expanded to some minimal resolution (e.g., 8 by 8 pixels) prior to perusing the display list. A mechanism is provided for determining all the objects that overlap this expanded area.

4.5 Image Scrolling

A key problem is displaying large images or maps; much larger than one display screen can show at a time. In addition, the user wants to be able to scroll around on this image or map, and to zoom in to selected sections of the image or map.

The basic approach is to divide up the overall image into rectangles that can be moved from disk to memory and from one place in memory to another in a very short time (considerable less than a second). These image objects are a subclass of the image class, called tiles, and have the smaller container class instances that correspond to each of their parts. A slight overlap in tiles may be allowed to resolve border problems. When the user elects to scroll the image, i.e. move mouse, the pointer position is used to select the additional image tiles needed and to recover them from disk. Latency is addressed by mating the default tile size to the minimum block size for retreival and the bandwidth of the cpu to disk (or other storage) channel. Note that it is assumed that the environment can find out the size of an image in external storage. In that case, it can be tiled as it is read in and stored in the local database for use during the working session.

4.6 Menus

Most windows in the vision environment have a region where fixed menu selections are advertised, plus a region that displays the variable information displayed by this type of window. Within the display region, the menus are a function of what object is selected.

5.0 Databases

A database provides for the management of many types of persistent data including the objects being operated on, the functions in the function library and the versions of each of these items. Management means keeping track of the items as well as storing or saving them from one execution of the program to the next (persistence). A single database (consisting of multiple files) is associated with each workstation. Multiple workstations in a cooperative environment require either a shared database or a distributed database approach (probably the latter).

This database accomplishes many different functions including:

1- simple persistent storage of images, objects, functions and other data,

2- flexible conditional queries to retrieve or compute instances of objects,

3- structural ordering of the object instances to provide fast, efficient access to the objects,

4- a consistent convention for accessing a variety of different objects with a minimum of coding effort, and

5- extensibility to support group data sharing on different physical databases.

The database manager is organized in a client-server model and consists of two components:

1- The database interface (or client) provides the interface to the database from any other programs. This interface is provided as methods that the other objects may invoke. Such methods include: insert, delete, save, find, etc.

2- The database server manages the storage and access to items assigned to the database including the allocation and deallocation of space. This includes creation, documentation, modification, access, and deletion of user objects (images, features, etc.) and programming objects (functions, documentation, numbers, characters, etc.).

Databases traditionally provide a shared access that prevents two users from interfering with each other when accessing the same object and provide a transaction system to ensure the integrity of each interaction with the database. These aspects of a database are not as important for the IU environment and are not discussed further here.

5.1 General Database Approach

The approach to the database design is in two levels. These are:

1- The basic level provides for the storage of objects, groups of objects and indices as files for management by the operating system. These files can be stored and read directly from the program or under interactive user control.

2- The next level provides interface to database management systems from commercial products. The current plan is to develop a generic SQL interface for the class hierarchy. This allows interaction with standard relational database system (e.g. Sybase, Ingress, Oracle). Interface to or new object oriented databases (Ontologics' Ontos) is a future possibility. Hooks are provided to build additional structures for efficient access, such as quadtrees.

The database is integrated into the core workstation software in several ways:

1- The primary access to the database is through the C++ language. The user (developer) takes advantage of the database through the core object structure.

2- Our approach to the database uses the strong typing feature of C++ by allowing the database objects to be incorporated directly in the object hierarchy transparently. That is, the objects are compiled directly into the program (with strong type checking) and are stored in the database by invoking the persistence attribute.

3- All imagery, imagery objects, functions, production rules, reasoning structures, etc. are expected to be stored in the database and access through the same C++ program interface

4- A set of user interface procedures that form a front end to the objects stored in the database.are provided for the developer

5.2 Database Implementation

The Sybase relational database system provides the basic relational database capability. An object oriented view of the database is provided by a set of object oriented procedures used as a front end to the relational database and the objects are stored using some object oriented structures build on top of Sybase.

5.3 Object Storage and Retrieval

Objects that are capable of being stored in the database are given the attribute persistence (inherited from a high level object). When this attribute is turned on, the object is guaranteed to be stored in the database for later access. It is up to the application developer to assign the appropriate attributes to the object so that it may be accessed properly.

This approach assumes that the structure of objects stored in the database is known to the compiler; in fact, the object storage mechanism is compiled and linked with the application program. It also assumes that the persistent objects have a standard set of methods for storing, retrieving, inserting, ordering, etc. These methods constitute the interface to the database and must be chosen carefully to allow replacement of the database structure at a later date.

One function of the database is to provide a s mple way to keep track of the images, features and other objects in the system. Each object is assigned a set of identification information, source, dates and data characteristics. This information is stored along with the corresponding object. The database allows users to access and

keep track of all images and to select subsets of these images according to various selection criteria for viewing or processing.

The index or ordering information on objects in the database is provided by special objects that have the appropriate structures. Any group of objects may be ordered using this object type by creating an appropriate indexing object. These indexing objects include: binary trees, quad trees, oct trees, hash tables, etc. The indexing objects access the ordered objects by providing an offset into a table storing the data for these objects.

Another function of the data base is to store information derived from the objects. These are arbitrary sized objects and may be retrieved by a variety of attributes. The attributes of these objects are defined by methods on the objects. These methods/attributes may be precomputed and stored in some kind of indexing list or they may be computed when the query is made.

5.4 Processing History

Another database function is to store the history of processing performed on the various image objects. This capability is similar to the source code control system used by software developers. The history system maintains a complete version of the image object along with enough information to reconstruct any other versions of the image object. The processing performed to extract any information (features, etc.) is recorded so that it may be reconstructed if desired. This capability allows the user to use the data from a previous step and to try alternative processing sequences to arrive at new conclusions. All such processing paths are recorded.

The history management system records the sequence of operations that are performed on each image. The original image (or image object object) is stored and the commands with relevant parameters are recorded for each sequence of operations. This is sufficient to recreate the results of any processing sequence. In addition, the intermediate results may be stored, if the user elects to do this.

6.0 Code Libraries

Libraries that support model-based reasoning and signal/image interpretation are built to manipulate the core objects described in section 3. The interface to each function is composed of core objects as far as possible, and the simplest and most general core object as possible to maximize reusability. For example, a two-dimensional window-based transform is designed to operate on a stream of linear arrays. Then any regular two-dimensional container, whether it is embedded in 2-, 3-, or N-space, can be accessed as a stream of linear arrays, and piped into the transform. The core objects support any reasonable data conversions, with priority given to conversions that can be done by a "forgetting" mechanism (e.g. forgetting a linear collection of integers was constructed as type 3d array).

Libraries are themselves a hierarchy of sub-libraries, allowing the incremental inclusion of software packages into applications. The set of possible functions have been grouped into five top-level categories:

Sensor and Image Processing Library
Model Library
Matching and Grouping Library
Interpretation Library
Reasoning Control.

Each library at this level corresponds to a processing stage and/or subsystem in an IU or decision support system. The sensor and image processing library is a collection of objects and routines that process data received from an external source, emulating the interaction of sensors with the objects defined in the input data and/or applying various filters and feature extraction. The model library consists of objects and routines for modeling the world spatially, temporally, and in any other relevant qualitative or quantitative domain. The matching library comprises routines for matching a model-based prediction of the world to sensor-derived evidence of the world.

The interpretation library is a collection of approaches for classifying objects that have been isolated from sensor input and/or mechanisms for inferring decisions. The reasoning control library is a collection of techniques for controlling the execution of this classification/inference process.

6.1 Sensor and Image Processing Library

These are image and signal processing routines and sensor modeling objects. They can be categorized as follows:

Sensor Modeling Objects
Pre-processing (spatial transformations)
Segmentation (extraction of higher-level structures)
Feature Calculation

Sensor modeling objects emulate sensors in predicting the interaction of energy with objects representing matter and energy in the world. The pre-processing functions are operations on 1, 2, and 3-dimensional collections of sensor values that

transform, but typically do not reduce the information content of the data (e.g. look-up-tables). Segmentation involves extracting higher-level structure from this data, such as extracting 2d curves and surfaces from a 2d image. Feature calculation computes statistics on these higher-level structures, resulting in collections of measures.

Appendix B lists the public-domain source packages we have identified for further consideration for inclusion in the environment. We also list source packages that are available for one-time license fee with no royalties on reuse of object code.

6.2 Model Library

This library supports the representation and manipulation of spatial, temporal, and other models. Spatial models are typically 2 or 3-dimensional objects, represented as a discrete collection of pixels (or voxels) or represented symbolically with an equation and/or algorithm that describes a shape. Connected models are built out of simple models, linked with constrained coordinate transformations (through methods of coordinate objects) that define the degrees of freedom between the two parts.

The library supports the projection of these models onto other coordinate systems and lower dimensions. 3d models are projected onto a 2d screen. The library also supports the symbolic manipulation of symbolically-represented models, in order to project the model's 3d constraints into a predicted 2d view.

6.3 Matching . and Grouping Library

This library consists of routines for placing higher-level interpretations on information extracted from sensor data, by exploiting information stored in models, and matching what is known about the models to what has been observed in the sensor data.

Predictions of what might be seen in the sensor-data constrain interpretation of the sensor data, and, conversely, evidence accrued from the sensor data, directs the consideration of possibilities in the model. Perceptual grouping is included in this library, because it relies on a priori facts about world structure in order to decide how to perceive sensor data.

6.4 Interpretation Library

This library supports various approaches for machine inference. Typically the inference to be made is the classification of some object that has been pre-processed by the sensor and image processing, modeling, and matching/grouping libraries. This library supports multiple inferencing approaches, including statistical classifiers, bayesian inference, logic engines and neural nets.

6.5 Reasoning Control

This library is the control layer for the inferencing that is represented by the interpretation library. It decides what (abstract) processing task to do next, using a particular strategy such as weighing the expected cost, the expected value of information, etc.. and using this metric to choose the next best task to attempt.

Note that reasoning control is not the same thing as process control described below. Process control is a mechanism for sequencing processes using standard

programming techniques, and has no special knowledge of decision making techniques.

7.0 Implementation Issues

This section describes the software development and test plan including the management procedures in Section 7.1. Section 7.2 discusses some of the implementation specific issues, that is, issues that are specific to a particular choice of hardware or support software package.

The order of implementation is to produce some core objects and their display methods first. Polygon objects (1,2, and 3D) are the first core objects implemented. Then, the following may be performed in parallel with appropriate interactions to provide a consistent system.

1- User Interface
2- Function/Object Libraries
3- Inferencing
4- Database System

The run-time environment grow sout of the user interface and is developed after the remaining functional areas are essentially in place. This is a recursive/parallel approach in that the four functional areas can be done essentially in parallel with each area being recursively developed to provide additional capabilities. Several design issues that impact the efficiency of both the design/build process and the resultant environment are addressed here.

7.1 Software Management Procedures and Tools

7.1.1 Development Code and Document Version Control

The code and documents for this project are managed using the RCS code control system. This means that all versions are recorded and can be recovered. It also means that the object codes that are created can be strictly controlled. This is important for achieving reusability of code.

7.1.2 Library Maintenance and Installation

The key issues are standards for adding code to libraries, and methods of adding new (sub-) libraries. Standards are addressed primarily by documentation and version control.

The basic method of adding a (sub)library is to extend the class hierarchy to create objects that appropriately utilize new packages of methods. So, for example, multiresolution pyramids are a subclass of connected objects. multiresolution search methods can be added to the objects, or stored in a new sub-library of the matching and grouping library that deals with multiresolution structures.

To aid installation of new libraries, graphical, table-based installation of graphical programming options (e.g. new object creation menu) are provided. In addition, how-to source-code examples, an installation manual, and appropriate make files are provided.

7.1.3 Testing Approach

A public-domain test interpreter, icp, can be applied to each C++ module to test its functionality. Manual code walkthrough after development and initial testing looks for exceptions such as poor memory allocation, inappropriate function calls, missing methods, and unexpected interactions between objects. Integrity testing will be built in to a limited extent, such as code within an object that checks internal pointers for consistency, or writing a known pattern in "dead space" at the end of data structures to catch overwriting.

7.2 Windows Interface

The Xwindows interface is through the InterViews package. This package provides a C++ toolkit for using Xwindows.

7.3 External Data Base Interfaces

There are three aspects to an external database interface: formulating a query that is understood by the serving, external database, communicating with the database to input the query and receive the returned data, and third, handling the returned data.

Most external databases of interest at this time are relational databases that speak standard query language (SQL), such as Sybase, Oracle and Ingress. A graphical interface is formulated that represents data objects as icons or text, and uses graphical selection methods to indicate ANDs, ORs and NOTs, such as pluses between icons for ANDs, an oval enclosing multiple icons for ORs, and the classic diagonal red line for NOT. The graphical query is translated to SQL. This approach gives a generic SQL interface.

The interprocess communication between the serving database and the client IU environment is managed by internet and Unix protocols, with the IU environment running the, possibly networked, database query in the background.

Large volume data returns are handled by two methods: a trap at the internet protocol level to threshold the amount of data that the system is willing to receive (this level can be user set), and a paging/buffering scheme to handle large images and large numbers of images.

7.4 Efficiency Approaches

Operations within the vision environment can be accelerated by three methods: addition of various hardware accelerators, multiple processes communicating via shared memory and other Unix inter-process communication mechanisms, and the networking of multiple platforms. Each of these are discussed in the following subsections.

7.4.1 Hardware Accelerators

The obvious accelerators are relatively inexpensive boards that are Sun compatible, such as the one provided by Vitek. These boards are treated as internal compute-servers, and interaction with them is via a prescribed set of function calls. The

function calls accomplish both the download/upload of data, and the setup and invocation of computation.

7.4.2 Shared Memory

Multiple processes can be used to accelerate operations by allowing a compute-intensive operation to be spawned as a background process (or be put in the background automagically when the user begins interacting), leaving the foreground processing available for user interaction. In addition, the time spent waiting for disk I/O and other hardware interactions can be reapplied to other processes.

7.4.3 Networked Platforms

An alternate to add-in boards is a network of machines, some of which act as compute-servers. These machines are dealt with in much the same manner as the add-in boards (see section 7.4.1), with the added possibility of file sharing through use of the Network File System (NFS) standard. In this case, the network is being used to farm out specific computational chunks, rather than being a full distributed problem decomposition. One possible implementation is by remote procedure calls (RPC)s as background processes.

8.0 Applications

8.1 System Engineering Application Development Approach

A high-level approach to generating requirements for the core and domain specific workstations (ws) is shown in Figure 8. Each domain area defines a set of tasks that correspond to client solutions to representative domain specific problems. Requirements are generated specific to each task that would result in a solution if built to spec. These are sifted together to take our best cut at the common core requirements. The generic, core ws is designed to meet these requirements. Based on the core design, extensions are designed to fulfill the domain specific requirements for each domain task. Of course, domain ws should strive to maximize synergy between solutions to multiple problems in the same domain (instead of developing many diverse ws to fulfill diverse tasks). Some of these steps, especially the last three, can be performed in parallel, but the sequential flow picture captures the philosophy of the approach.

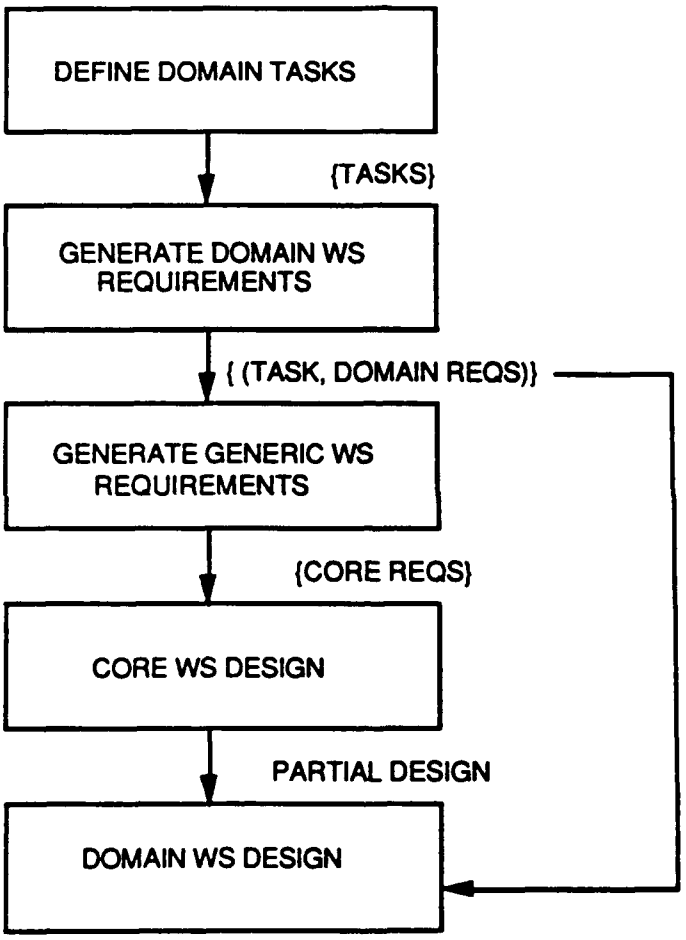


Figure 8: Approach to Requirements

The objectives of an IU Workstation are to provide a hardware and software environment that greatly facilitates development of IU application systems by providing a core infrastructure of integrated tools that are common to most applications, and that traditionally take up the lion's share of development time when systems are built from scratch.

We believe the most direct path to defining such an IU environment is to define domain specific IU task applications and design the IU workstation environment so that it supports development of systems that can perform the domain tasks. In other words, the IU environment is used to build systems that then solve domain problems, but we begin with the domain problems so that we are sure the resulting environment can solve those problems. In order to guarantee building an IU workstation environment with suffcient generality to apply to IU development in many domains, we have adopted the approach pictured in figure 8 to generating requirements for the IU workstation.

Diverse domain areas are chosen and representative tasks are defined in each domain area. Requirements are generated specific to each task that would result in a solution if built to spec. These are sifted together to take the best cut at the common IU workstation requirements. The generic, core IU workstation is designed to meet these requirements. Based on the core design, extensions are designed to fulfill the domain specific requirements for each domain task. We have selected two domain areas, terrain analysis and medical imagery analysis, to focus our workstation development.

8.2 Domain Task Analyses

As explained in section 8.1, two task areas, terrain and medical IU applications, were selected as foci to guide workstation development. The terrain task analysis is presented in section 8.2.1 Section 8.2.2 presents the medical imagery task analysis.

8.2.1 Terrain Task Analysis

We have selected two terrain domain tasks: semi-automated, syntactic, "snap-to" digitization of hardcopy maps and interactive, semantic map/image measurement tools. The terms used in these task descriptions deserve some explanation. "Semi-automated, syntactic 'snap-to' digitization" is the task of interactively choosing a small set of points on a terrain feature from a bit-mapped image of a scanned (i.e., a "digitized" image in the conventional sense) hardcopy map and having the workstation infer the full set of points on the feature (a road, for example), but without specialized knowledge about the feature, except perhaps for basic geometric properties (e.g. linear feature, area, etc.).

An "interactive, semantic map/image measurement tool kit" is a set of measures that can be used on already digitized maps (i.e., scanned and interpreted maps, stored in a digital terrain database), but can also be used at the time of digital terrain database creation to compute and store
attributes. Potential tools include line of sight calculations, earth/hole volumes, shortest-path computation, time-to-travel-path, 3D view visualization, etc.

Roads are a good first focus for semi-automated map extraction, both because they are one of the simplest map features to define for machine segmentation, and because they are required for many common applications. Similarly, line-of-sight analysis is a good first choice for measurements, because it is commonly used, and because it

requires manipulation of elevation maps, which is technically difficult to do without computer aid.

Examples of features stored in digital terrain databases and associated textual (usually relational) databases that are candidates for semi-automated extraction include roads, railroads, hydrology features, land parcels, agricultural plots, fields, forests, task-defined elevation features (e.g. hills, air-space obstructions), and task-defined sets of buildings (e.g. residences, industries). Examples of measurements made from digital terrain databases include, line-of-sight, land volume, areas of different (and possible compound) terrain types, watershed, distances between points and between geometrically more complex terrain features, such as rivers and towns, as well as topological relations for different terrain features including between, nearness and adjacency.

The objective of terrain task analysis is to decompose the execution of the terrain tasks such that the underlying required functionality is naturally exposed, making more obvious the design solutions to building a system that achieves this functionality. Our approach to this is twofold: first to step through the system user's tasks in scripts, essentially scripting (or verbally storyboarding) the functionality of the user-interface, and second to analyze data transformations as the user experiences them (based on the storyboard) inferring progressively finer levels of black-box functionality the system must possess. The terrain task script is presented in section 8.2.1.1, and the user-apparent data transformations are discussed in section 8.2.1.2.

8.2.1.1 Terrain Task Script

1) Select map, scan (i.e. bitmap digitize) if necessary.
Assumes: Maps exist in an on-line accessible form, and/or there is a user-friendly scanning procedure available for hard-copy maps. The latter also assumes that scanning resolution is sufficient versus map detail to enable the feature selection, etc. of the following steps.

2) Indicate feature type (e.g., linear, area, road, field...) or measure type (e.g. line-or-sight, shovel, hourglass, spyglass...).
Assumes: Feature and/or measure types are relevant to map features and to the domain task (e.g. urban planning, agricultural survey, etc.)

3) Select feature or measure type.

4a) Select mode to label feature and type-input for feature label.

5a) Choose points on features in displayed map.

6a) Select mode to do snap-to feature segmentation or to perform the measure.

7a) View the displayed results of feature extraction and/or measurement.

8a) Modify the displayed results as necessary.

9a) Select mode to store labeled features or measures to database.

Assumes: Storage mode is a known default. Otherwise this step must be enhanced to select storage location. Typically, the map and/or measure type will have a storage location and format associated to it already.

4b) Select choose feature mode and type-input for label for the feature.

5b) Choose points on features in displayed map.

6b) Repeat 4b and 5b until all labeled features selected for extraction or measurement.

7b) Select batch-mode for snap-to feature segmentation or measurement.

8b) Select mode for viewing results of batch feature extraction or measurement.
Assumes: Job is identified by user or other method and accessible by that identification. If user has multiple jobs, system must appropriately differentiate and allow user selection for viewing.

9b) Modify the displayed results as necessary.
Assumes: All results of this job are simultaneously displayable and usefully viewable (e.g. non-overlapping). If not, steps 9-10 must step through feature by feature or screen by screen.

10b) Select mode to store labeled features or measures to database.
Assumes: Storage mode is a known default. Otherwise this step must be enhanced to select storage location. Typically, the map and/or measure type will have a storage location and format associated to it already.

8.2.1.2 Terrain Task Data Transformations

A terrain task data transformation refers to the displayed output a user observes in response to a set of inputs while performing the terrain task script. Inputs include items selected or typed by the user, as well as data the user assumes is present, such as scanned maps and digital terrain databases. The "transformed data" includes the visually observed displays, such as a bit-map overlay of an extracted terrain feature, as well as the implied data the user assumes supports the display, such as the bounding polygon of the segmentation. The user-apparent data transformations suggested by the task script include the following.

1) selected and typed entries -----> map (= scanned map image + any associated textual data)
2) selected map -----> displayed map and textual data
3) selected feature type and chosen points -----> segmented feature
4) feature segmentation -----> feature segmentation (interactive)
5) selected feature type and chosen points -----> segmented feature + measure
6) selected extracted features and/or measures + labels-----> database record (indicating storage of features and/or measures)

The transformations imply the existence of certain user-apparent system functionalities. These are listed below.

Transformations (1) and (2)

A database(-like) facility is implied that consists of a set of storage units for scanned maps imagery and associated textual data, with the database facility inverted on one or more of

a) map type
b) measurement task type
c) map source
d) digital database for map feature and/or measurement storage
e) job identifier

For single-task users, like urban planners, the measurement task type or map type are more likely keys. For multi-task users, like DIA, or a centralized city GIS facility, the map source and/or storage database are more likely keys. In the former case, the user usually accesses the same set of maps, extracting information and making measurements from it. In the latter case, the user accesses from a wide variety of maps and map databases, often updating the maps and associated databases. Job identifiers can be used to allow individual users or teams to work incrementally on an on-going task. A job has an associated data structure that saves pointers to the input and output data sources and job state. They can be indexed by user names or task names.

Transformation (3)

This is the critical technical step of semi-automated segmentation. The feature extraction itself implies a specific choice of segmentation methodology, and the use of specific image processing, pattern recognition, search, and inference procedures. The specific choices are not user-apparent. However, the output display is user apparent and must allow the user to "verify at a glance" the correctness of the terrain feature extraction. This probably requires a side-by-side display of the system segmentation in graphic overlay, next to the un-segmented scanned map.

The choosing of some points on the terrain feature in the displayed map serves the two purposes of indicating which feature is being stored and associated to the typed-in label, and for seeding the otherwise automated feature extraction process. For example, to semi-automate road segmentation in a scanned map, the user could be directed to choose the two endpoints of the road segment, then the system extracts the road segment between the two points. The choice of points clearly depends on the feature extraction task; the user must be directed how to choose points so that they are meaningful to the system with respect to the user's task. On-line instruction regarding geo-object choices for the various feature types and data sources should be available.

Transformation (4)

The user views the system's display of its terrain feature segmentation results and allows the user to interactively edit the results if necessary. This is done using the operations add, delete, move, undo and save, as defined in the glossary. The regions and/or geo-objects should snap-to as points or other geo-objects are added or deleted. A reasonable first definition of snap-to is gotten by using shortest-planar or shortest-elevation-grid distance to link points or other geo-objects.

Transformation (5)

A measure requires dynamic extraction of terrain features when they are not pre-stored. In either case the system must understand a priori what features are required. The points chosen by the user define the geographic region of interest for the measure. If feature extraction is necessary, the user should be cued appropriately. The system should be set up so that it is easy for the user to abort the process if it requires feature extraction s/he does not wish to do.

Measures can output complex data structures such as geo-objects and/or regions, as well as textual and numeric outputs. For example, line-of-sight calculations output a polygon or elevation region with line-or-sight from a point or other region. Land volume, on the other hand, outputs only a single number. Some measures require sohpisticated geometric computations, such as surface, polygon, and/or line intersection, union and difference. A full polygon algebra should probably be supported.

Transformation (6)

Status messages for intialization and completion of tasks should be displayed. Storage and retrieval of large volumes of data should have displayed meters or other devices to indicate that the system is engaged in a task and how close to completion it is. Storage and retrieval times should be estimated and warnings issued for time consuming processes allowing the user option to alter or cancel the storage or retrieval command. Asynchronous, batch-mode storage and retrieval should be supported.

Systems that provide multiple users to cooperate on the same task may have special requirements for data integrity, such as write-lockouts to avoid synchronous map updating.

8.2.1.3 Terrain Applications Requirements

In both terrain tasks, the top level user interaction steps are as follows:

1) Select maps/imagery, scan (i.e. bitmap digitize) if necessary.
2) Indicate feature type (e.g., linear, area, road, field...) or measure type (e.g. line-or-sight, shovel, hourglass, spyglass...).
3) Select points on features in map and/or image.
4) Tell system to input features to database or to perform the measure.
5) Verify correctness of feature extraction and/or measurement.

Infrastructure requirements such as friendly and efficient user interfaces, a need to robustly interact with digital terrain databases, large data storage and retrieval capabilities, etc. are common to more than one step. Nonetheless, in the following, we attempt to list requirements in correspondence to the user functions, rather than in terms of elements of the solution (like user interfaces).

1) Select maps/imagery, scan (i.e. bitmap digitize) if necessary.

1.1) The system must be equipped with standard map scanning (e.g. flatbed digitizer) interfaces that provide adequate resolution to capture the necessary map detail.

1.2) If scanner is used, then digitizing should be done on screen on the scanned map.

1.3) Sufficient workstation memory is required to hold the terrain database corresponding to the scanned map segment in memory.

2) Indicate feature type.

2.1) The system must interface to ARC/INFO, and a selection of standard terrain digital databases to be determined by the target market.

2.2) We should choose and make available a digital terrain database that comes standard with the product. Coverage and resolution requirements will be determined by market analysis.

2.3) A uniform interface should be available for all terrain databases the system communicates with.

2.4) A display showing the selectable terrain features present in the database corresponding to the displayed map segment should display in less than 10 seconds from the time map coordinates are entered, if the database segment is in local storage.

2.5) If the digital terrain database segment requires more than 15 seconds for retrieval to display, a message should be displayed telling the user the function that is being performed, and giving dynamic indications of progress.

2.6) The user should be able to select the digital terrain features with a minimum of manipulation (e.g. keystrokes, mouse clicks, etc.).

3) Select points on features in map and/or image.

3.1) It should be clear how to enter the modes to indicate points.

3.2) The points should be coded so that it is clear which correspond, to which features and/or measures; it should be intuitive and require minimal manipulation to change points, "move" a point in either map or image, etc.

3.3) It should be obvious how to indicate that point selection is complete, either on a feature, or on the entire image/map.

4) Tell system to input features to database or to perform measures.

4.1) It should be obvious how to indicate that point selection is complete.

4.2) The system should return control to the user in less than 10 seconds; either input and/or measurement should be available for verification or relegated to an off-line or background process.

4.3) If producing the verification display needs to be a background process, then there must be a protocol of queuing and recalling verification displays at a later time (e.g., stored up in a pull-down list). Any job that has been completed as far as the user is concerned should appear in this list, with its status indicated even if it's not yet available for verification.

4.4) If any process that occupies the user interface (so that the user cannot get response from the workstation) takes more than 10 seconds, a message should be displayed indicating operations in process.

5) Verify correctness of feature extraction and/or registration.

5.1) A verification display should be available showing an overlay of extracted features against map and/or imagery features. This should be togglable so that the user can switch between the original and extracted in focus of attention.

5.2) It should be possible to interactively correct the extracted feature; restorage must adhere to the requirements in (4).

5.3) Correction of feature extraction should be intuitive; it shouldn't be necessary to read a manual.

8.2.2 Medical Task Analysis

We have selected two tasks: automated intercortical volumes from hand radiographs and automated prostate volume quantification in CT imagery. Both measures are key medical indicators; the first in diagnosing and tracking arthritis, the second is a direct indicator for prostate surgery (radical prostatectomy).

The choice of the arthritis measure application is motivated by the following facts. The physics of sensor interaction of radiographs with human hands is a completely modeled, well-understood technology. Probability models for scattering of xrays in soft-tissue, bone, and air are commonly available in the medical physics literature [Curry et. al.-84], [Kereiakes et. al. -86]. This forms the basis for strong prior probabilities in imaging models. In the application of radiographs, the imaging geometry, approximate object (i. e., hand) aspect, and the ambient characteristics of the energy source (i.e., xray voltage) is always known, so this is a highly constrained, but not a toy problem. In this respect it is representative of a broad class of medical, manufacturing and inspection tasks for machine vision. Finally, hands are complex enough that the modeling problem is important, but simple enough that we can hope to accomplish the task within a reasonable project scope. Hands are 3D with articulated joints. Because the sensor is invasive, it is necessary to model the layered volume (not just the surface). Most of the primitive hand components are cylindrical in basic shape [Meschan-75]. We anticipate that it is not necessary to model deformable surfaces for hand recognition. Some population statistics for normal variation in bone size and range of joint articulation are available in the medical literature [Poznanski-74].

Prostate volume measurement requires working with 3D CT imagery, requiring image processing operations to exist for 3D image processing, and similarly requiring surface matching and volumetric understanding as is required in range imagery. The prostate requires irregular curved surface matching, but having genus one, is much simpler than the heart, for example. Additionally, the prostate is a static organ, unlike the heart, so that shape remains stable over time.

As in the terrain task, the objective of medical task analysis is to decompose the execution of the medical tasks such that the underlying required functionality is naturally exposed, making more obvious the design solutions to building a system that achieves this functionality. We again present a task script, section 2.2.1, followed by the medical task data transformations in section 2.2.2.

8.2.2.1 Medical Task Script

1) Order exams/measures
    1.1) Select order mode
    1.2) Select appropriate fields (e.g. modalities, views, measures)
    1.3) Type input for name, institution id OR use OCR to scan patient demographic data

1.4) Select results viewing station(s)
1.5) Select execute mode

2) Call up results
2.1) Select results mode
2.2a) Type input or select "my" icon to call up batched exams
Assumes: batched exams and "my icon" exists
2.2b) Select appropriate fields
2.3b) Type input for name, institution id OR use OCR to scan patient demographic data

3) View results
3.1) Select exam(s) to view
3.2) Select retreive and/or display mode

4) Modify results
4.1) Select modification mode
4.2) Select sub-mode of delete OR add OR move OR undo
Assumes: modification mode selected was "segmentation"
4.3) Choose geo-objects
4.4) Select done-modify

5) Order additional measures
5.1) Select measures mode
5.2) Select appropriate fields
5.3) Type input as required by 5.2 (typically none required)
5.4) Select results viewing station(s) (should default to current settings)
5.5) Select execute mode

6) Archive results
6.1) Choose exams to archive
6.2) Select archive mode
6.3) Choose archival devices
Assumes: More than one archive available
6.4) Select execute mode

8.2.2.2 Medical Task Data Transformations

A medical task data transformation refers to the displayed output a user observes in response to a set of inputs while performing the steps in the medical task script. Inputs include items selected or typed by the user, as well as data the user assumes is present, such as medical imagery from exams, and demographic data routinely associated with the exams. The "transformed data" includes the visually observed displays, such as a bit-map overlay of a segmentation, as well as the implied data the user assumes supports the display, such as the bounding polygon of the segmentation. The user-apparent data transformations suggested by the script include the following.

1) selected and typed entries -----> exams (= imagery + textual data)
2) selected exams -----> displayed imagery and textual data
3) selected and/or displayed exams -----> segmented imagery
4) segmentation -----> segmentation (interactive)
5) selected exam -----> measure
6) two selected exams -----> comparison measure

The transformations imply the existence of certain user-apparent system functionalities. These are listed below.

Transformations (1) and (2)

A database(-like) facility is implied that consists of a set of storage units for exam imagery and textual data, with the database facility inverted on

a) patient names by lexicographical ordering
b) patient by social security number
c) patient by local institution id
d) anatomy by hierarchical anatomical structuring order
e) dates by chronological ordering
f) modalities by lexicographical ordering OR data storage source
g) non-local institutions by lexicographical ordering.

The options are in priority order for the medical task. Most access is concerned with specific individuals, however, doctors also need to be able to access on anatomy and modality for purposes of teaching, research and demonstration. The options (e) and (f) are probably not both necessary to invert on (because the bucket size of retrieved exams will be small after patient and/or anatomy and/or either one of modality or date are selected), but are included for completeness. The option (g) will only be applicable when multi-site institutions are involved, such as the VA, Humana and Kaiser medical centers, hospital networks in socialistic countries (as most are run in Europe, for example), DoD hospitals, distributed clinics, etc.

Because of the large size of imagery in exams (typically from 6 to 100 megabytes per exam), it is standard practice to separate the textual data from the imagery, and to make it available from database queries without requiring associated imagery retrieval. The non-imagery exam data includes all textual and numeric patient data, including demographic data, dates and locations of visits, attending and referring physicians, modalties of imagery, anatomy imaged, imagery numbers, sizes, bit-depths, and pointers, diagnoses, measures, and additional physician's comments.

Transformation (3)

The capability to segment imagery implies a specific choice of segmentation methodology, and the use of specific image processing, pattern recognition, search, and inference procedures. The specific choices are not user-apparent. However, the output display is user apparent and must allow the doctor to "verify at a glance" the correctness of the segmentation. (FYI, doctors often call segmentation "contouring" imaged tissues. Bones, organs, and other soft tissues like ligaments, fat, etc. all fall within the taxonomic category called "tissues".)

Transformation (4)

The user (a physician or highly skilled technician) must be able to work directly on the verify-at-a-glance displayed output of transformation (3) to modify it for any system created segmentation errors. The operations of add, delete, move, undo and save are the tools for this. The user should be able to push the display around until s/he likes what s/he sees, and then save the result, which will be used for auto-recalculation of diagnostic measures that depend upon the associated segmentation.

Transformations (5) and (6)

The basic measures are area and volume of segmented tissues. For an area, this is computed as pixel count multiplied by the appropriate (constant per image) factor that transforms pixels to square centimeters based on the viewing geometry and physics of the scanner. For computed radiography (digital xray) we can automagically measure this factor from the grid that appears on the borders of the image. For digitized imagery, we must input this factor based on specific scanners and viewing procedures, or we must automagically measure it from a marker of known size placed in the image. The standard such marker is an "L" or "R" that is routinely placed on most xray imaging plates to indicate the view the patient was imaged from.

Other associated area measures include longitudinal axis computations for phalanges, average pixel intensity, maximum region diameter, and specific region diameters that depend upon finding certain anatomical points in the bounding polygon of the segmented region.

Volumes are usually computed by doing the areas of the slices, and then interpolating between adjacent segmentations. The interpolation is straightforward, but depends upon knowing the thickness of, and distance between, slices in the exam (as the set of images making up the volume is called). Usually these are constant factors for a given exam, but they do not have to be(!).

Comparison measures usually compare the measures the last time the patient was in against the current measures. So this requires the system to call up the last exam, make the measures if they weren't made then, and then also to do the measures on the current exam. Comparisons will typically be presented as percent increase or decrease, as well as absolute increase or decrease. If there is a longer history, another presentation, such as a graph, might be nice. Physicians do not currently produce such graphical aids. Comparison exams should be presented side by side, with the old exam to the left of the new one.

8.2.2.3 Medical Applications Requirements

The medical workstation design requirements are driven by the workstation task requirements, and divided according to a top level breakdown of functional units including:

- Physician/Technician User Interface
- Network to Scanners, PACS, RIS, and HIS
- Databases: Anatomical Models/Scanner Models/Patient Exams/Imagery Features
- Image Processing Functions
- Inference for Anatomical Segmentation
- System Control.

In the following, we break out requirements according to this functional system decomposition.

1.0) Physician/Technician User Interface

1.1) The interface must be highly reliable and fault-tolerant. In particular, it should be possible to move between states without rolling back through each intermediate state, and to alter inputs and/or abort at almost any point, saving state(s) and without catastrophic consequences.

1.2) Manipulation (typing, pointing, etc.) should be minimal to accomplish any task.

1.3) There should ideally be only one visual focus of attention at any time on the screen.

1.4) It must be possible to paint an image in less than 5 wall-clock seconds. In general, no display should be slower than this (graphs, spreadsheets, etc.)

1.5) Displays must support a minimum of 16 bit deep pixels.

1.6) Full color overlays must be provided. Color should be 24 bits deep (i.e. 8 bits each of red, green and blue) and color-tables should be changable in software.

1.7) It should be possible to draw on an image, then erase the overlay without (apparently) affecting the image underneath. In particular, if the image needs to be re-drawn, it must be transparent to the user.

1.8) Both interactive and automatic imagery color overlay capability should be provided. Color tables should be interactively changable.

1.9) Zooming should be accomplished in no more than 5 wall-clock seconds. Both zoom window and zoom around point should be supported. Zooming should zoom both image, overlays and any "chained" windows.

1.10) Both 2D and 3D imagery display should be provided. 3D imagery must support 512x512x256, 2D must support 4Kx4K imagery. Imagery must be viewable in its entirety (i.e., without scrolling) in unzoomed mode.

1.11) Volumetric imagery must be viewable from any perspective. 3D and 2D rotation and translation must be supported.

1.12) 2D cutaway views of 3D imagery at an arbitrary angle should be supported.

1.13) 2D imagery display should "fill" the display window.

1.14) 3D graphic object display should be supported including raster and vector representations with hidden line and surface capability.

1.15) It must be possible to overlay 3D graphics on 3D imagery, with the same erasability requirements (see 1.7 and 1.8) as 2D overlays.

1.16) Display should convey the same information to colorblind people as it does to normal color-sighted people.

1.17) Any operation requiring more than 5 seconds of wall-clock time to execute should display messages indicated what it's doing.

1.18) Any operation requiring more than 15 seconds of wall-clock time should be provided in background mode so that other workstation interaction can go on.

1.19) Imagery displays should support mouse, light-pen or other pointer interaction.

1.20) Menus, sliders, buttons, tables, graphs, icons and sprites should be supported both interactively and automatically.

1.21) Imagery browsing should be supported with programmable choice of "sub-sample" function for reducing imagery size for browse windows.

1.22) If multiple screens are required to browse through a single patient exam (which can be up to 256 images), screen switching should be extremely rapid, certainly under 5 seconds, hopefully under 3 seconds.

1.23) The user interface should be easy-to-learn, easy-to-recall and easy-to-use.

1.24) Custom interface selections (e.g. customized defaults, button locations, etc.) should be available and easy to set up and use.

1.25) An easy to use access-security system should be provided. This security system must allow programmability of selection of access to system functions/modes/devices/data that are permitted for varying levels of clearance.

1.26) The user interface to external databases, etc. should be represented in visual, medical-domain terms/appearance.

1.27) The user interface should be portable to (almost) any Unix box with sufficient memory. It should provide network server capability (a la X, NeWS, etc.)

1.28) On-line help should be available but not needed by the user.

2.0) Network to Scanners, PACS, RIS, and HIS

(PACS= Picture Archival and Communication System, RIS= Radiology Information System, HIS= Hospital Information System)

2.1) The workstation must support standard networks, (ethernet, fiber-optic ethernet, hipi, etc.) and standard network communication protocols including TCP/IP, ISO, IEEE, etc. Custom networks must be supported for critical vendors (probably PACS systems not known at this time, scanner vendors we contract with, etc.)

2.2) High-speed/high-bandwidth communication must be supported. Exact numbers are not yet known, but probably in the hundreds of megabytes per second range.

2.3) Must decode and read imagery formats including ACR/NEMA, ISO and IEEE as well as "proprietary" formats for CR/CT/MRI scanners made by GE, Siemens, Philips, Toshiba, Picker, and Diasonics at a minimum. Data structures, selections etc. should support addition of formats as required.

2.4) Data exchange between internal databases and institution databases must be supported including all leading commercial PACS, RIS and HIS systems.

2.5) We must provide accurate modification tracking of changes to medical records, including noting who made what changes when.

2.6) The possibility of distributed updating of records must be accounted for, either by some communication protocol between workstations, or by prohibiting it (e.g. data locking).

2.7) Access to records must be security controlled to meet legal, medical and institutional requirements for patient privacy and medical protocols.

2.8) The workstation should be able to obtain dynamic models of the contents of the accessed external databases such that if an unusually large retrieval is indicated, the retreival is confirmed before execution.

2.9) The workstation should provide a uniform interface to all imagery/records storage devices. Details should only be available for debugging problems; otherwise it should look like a single database.

2.10) If multiple workstations are networked in and/or between institution(s), they should communicate to effect transfer of data to meet scheduling needs for availability of records/imagery where and when required for operational needs of the institution(s).

3.0) Databases: Anatomical Models/Scanner Models/Patient Exams/Imagery Features

We first list requirements that are common between databases, then the requirements that are specific to each.

3.1) We need to be able to query over arbitrary "keys" or access slots.

3.2) We need to be able to construct arbitary boolean algebraic queries over multiple database keys.

3.3) The interface to database queries, whether they are databases resident in the workstation or external to it, should be intuitive and easy to use by anyone with a high-school education (e.g., you shouldn't have to know boolean algebra).

3.4) Database searches should be optimizable (i.e., programmable), to depend on the type and space/time characteristics of the data being searched for (e.g. graph search over anatomic models, spatially oriented search for perceptual grouping in imagery, etc.)

3.5) The anatomical models and scanner models databases must be persistent.

3.6) The patient exams database must be locally persistent until it can be verified that the records/imagery have been archived in the appropriate external database.

3.1) Anatomical Models Database

3.1.1) The database should support representation, storage and retreival of point, curve, surface and volumetric objects, including graphs and boolean combinations of them.

3.1.2) Complex relations such as articulation should be supported between various modeled anatomical parts.

3.1.3) Pointers to image processing and feature extraction operators should be supported.

3.1.4) Models should be indexed by patient demographics. In particular, ranges and probability distributions over them should be representable for all model features.

3.2) Scanner Models Database

3.2.1) Models should be available for computed radiography, digitized xray, computed tomography and magnetic resonance imagery. Scanner modeling includes procedures for predicting appearance of imaged anatomy based on imaging geometry and anatomical parts (as represented in the anatomical models database).

3.2.2) Output of applying a scanner model to an anatomical and imaging model should be an imagery appearance prediction that is represented to be usable by image processing and inference operators.

3.2.3) Distributions of prior probabilities of imagery appearance based on 3.2.2 should be provided in an inference readable format.

3.3) Patient Exam Database

3.3.1) Records selection must operate from all minimally sufficient queries.

3.3.2) Self-contradictory queries must be handled, either by making them impossible, by offering auto-ORing and confirmation to the user, or some other method(s).

3.3.3) Matching of input data such as patient names, social security or other id numbers, etc. should support partial matches, be case insensitive and accept all possible standard syntaxes (e.g. first name, M.I., last name versus
last name, first name, M.I., etc.)

3.3.4) Ideally, mispellings, homonyms, and any other known standard data entry problems should be handled subject to reasonable speed requirements (TBD).

3.3.5) It should be clear if a measure has already been performed, and it should be easy to indicate re-doing the measure.

3.3.6) Multiple values for the same measure should be maintained, and the differences accounted for (e.g., if a physician interactively modifies a segmentation and asks for a recalculation, but all on the same exam.)

3.3.7) It should be easy to retrieve and view prior exams, and indicate which ones should be used for comparison measures and/or trend tracking; again standard defaults should be available and easy to indicate.

3.3.8) Multiple exam trends should be plotted in a self-explanatory, easy to read fashion, with links to multiple records available for display of interactively selected comparisons.

3.3.9) Linked records, e.g. comparison measures across multiple exams, should be indicated so that they can be quickly recalled for viewing.

3.3.10) Sufficient local storage should be provided to buffer exams. Sizes of this buffer are not yet known, but will easily be in the hundreds of megabytes and is likely to be in the thousands of megabytes.

3.4) Imagery Feature Database

3.4.1) Dynamically created feature databases as the output of image processing operators should be supported.

3.4.2) Feature databases should be indexed by the imagery the features were extracted from.

3.4.3) The databases should be spatially hierarchically represented to support optimization of search for feature groupings.

3.4.4) Features should be linkable as instances of anatomical models that are linked, when this makes sense (e.g. spatial relationships between extracted bone surfaces).

3.4.5) Feature databases need only persist until imagery analysis is completed.

4) *Image Processing* Functions

4.1) A large library of standard image processing functions must be provided, including arbitrary kernel sized convolutions, where the convolution window allows arbitrary arithmetic and logical operations on the neighborhood.

4.2) Arbitrary sub-window and blotch (i.e. mask dependent) processing should be provided for all IP operations.

4.3) It should be selectable on all operators whether the output takes the form of images, lists, or a feature database, where these outputs make sense.

4.4) Operators should be able to accept the output of feature database queries as inputs.

5) Inference for Anatomical Segmentation

5.1) Full Bayesian inference over either continuous or discrete values must be supported.

5.2) Bayes nets must be dynamically instantiatable to correspond to instances of hypotheses of instantiated models.

5.3) It must be possible to query the state of any subset of the Bayes net; reasonable subsets (e.g. subtrees) should be efficiently searched.

5.4) The Bayes net must be a savable structure, but need not, in general, be persistent.

5.5) It should be possible to efficiently search the model space and feature space to perform matching, and to efficiently instantiate Bayes nodes based on the result of the matches.

5.6) Metrics used for matching should be programmable; in particular, sophisticated match metrics such as the Mahalanobus distance should be supported

6) System Control

6.1) A full utility theory over the Bayes nets should be provided.

6.2) It should be easy to select and change value functions over the anatomy of interest. The value functions should be linked to the anatomical model database, and should apply to arbitrary levels of hierarchy there (e.g. curves and surfaces as well as volumes).

6.3) Control should account for listening to the user while still maintaining good efficiency in computationally intense processing.

6.4) The system should understand when "batch" type processing is required and when realtime interaction is required. If it cannot provide the latter, it should inform the user and give appropriate options.

6.5) System control should exhibit a certain level of fault tolerance; in particular, self-diagnostics should be periodically run, and appropriate message displayed if servicing is needed.

8.3 IU Application Development Task Analysis

The objective of IU application development task analysis is to present the generic steps an application developer often repeats, and also to present the dependencies between steps such that the underlying requirements for tool capabilities are naturally exposed, making more obvious the design solutions to building tools and a tool using environment that achieves this capability. Our approach to this is to step through the tasks a "generic" IU application developer performs in scripts of progressively finer detail, essentially scripting (or verbally storyboarding) the functionality of the user-interface. Section 8.3.1 is a summary of tasks the developer embraces in building an image understanding (IU) application. In section 8.3.2 the application developer's script is presented.

8.3.1 Summary of Application Developer Tasks

It is problematic to present a script of the actions an application developer performs, as developers may permute the order and dependencies among operations in many ways as they incrementally interleave development steps such as image processing, model building and matching, for example. So the workstation designer needs to be especially wary of depending too directly on the folowing script as a specification for doing tool development. We can be sure that the functionality listed in this script is a strict subset of what is actually required. Nonetheless, the intention of this script is to cover the main steps and the obvious dependencies between steps in sufficient breadth and depth so that the workstation designer produces a useful environment even if s/he takes a narrow interpretation of the ripted capabilities.

A second complication is the need for the IU application developer to deal (more or less) directly with many of the objects that are created during development sessions. It is tricky to discuss the required functionality without suggested design solutions (e.g. object structures) to simplify the language. However, the attempt is made here to state the functionality without designing solutions to achieve it. As a consequence the script sometimes reads a bit more like a set of tool requirements than a script of actions.

The generic task the script is focused around is that of doing the development to automate an IU application. Basically, the developer (we interchange the terms developer and user from now on) wants to bring up some imagery, interactively play with it to make measurements and begin guessing what sort of imagery operators can

work on the data. Then s/he wants to string together a bunch of existing image processing (IP) operators, playing with parameters interactively to see what evidence is extracted from the imagery.

Then begins the task of model development. The models are geometric, material, constraints, and IP (or other) actions for evidence gathering. This requires interactive geometric modeling, integrating population statistics, setting up and experimenting with constraint equations, displaying lots of 2D and 3D geometric objects, and interactively editing to create model objects out of these pieces.

Development of matching operators typically descends into a full programming environment, writing matching routines dealing with the model and IP objects that have been defined. Some basic math packages for solving cubic spline equations are helpful here, and probably similar tools for various other parametrizations. Grouping is a type of hierarchical and/or adaptive combination of searching and matching features into more complex structures. Groupers can be looked at as complicated matching routines, for the sake of how they fit into a development environment. However, they tend to be different in that they can make extensive use of Powervision style image feature filtering over a database of imagery features to do their job. The need to have computationally intense numerical matching operations in a tight loop with database calls that invoke spatial searches for the data to be matched against is unique to computer vision grouping operations (to the best of my knowledge). The idea of implementing this as database filtering is actually a design approach rather than a required capability.

Now the IP operators, models and matching and grouping routines are integrated together in an inference framework of some sort. Three of the most common such frameworks in IU applications are Bayesian inference networks, blackboard-executed frame-based systems, and logical rulebases. In any of these paradigms, the developer defines model representations of (Bayes) nodes/frames/rules that point at the set of model-components that can be confused in recognition. Conditional and a priori probability distributions/confidences or weights of evidence must be defined by either LUT or parametrizations and put into the network model/frames/rules. Models for the IP operations should include expected time of execution as a function of sw/hw environment and relevant input data parameters such as imagery size. The Bayes net/blackboard/inference engine gets cranked manually until inference starts looking reasonable.

When the net seems to operate well with human control, the next step is to experiment with automated control regimes. For Bayes nets, values get assigned to the top level nodes of the Bayes net, and utility functions can be generated from the bayes net. The utility function assigns a number to each action that can be executed as a self-contained process that comes from the bayes net. These numbers can be used to rank the processes for operating system style control like FIFO and best-first. Alternatively, a full, decision theoretic control can be used, or even rules giving an exact specification of steps to be executed in the Bayes net. In blackboard systems and rulebased systems, meta-rules must be developed that guide search routines and prioritize rule execution.

Finally the whole thing's gotta be tested, a lot, by components and by system. Statistics from runs should be automagically accumulated. Timing and other standard software metrics should be provided at a tool level. Tables and graphs should be easy to generate. Then the whole schmeer has to be software engineered for maintenance, documentation and versioning.

8.3.2 Hierarchical Application Developer Task Script

The "script" detailing the above development task summary is presented in levels, each subsequent level expanding the detail of the previous. Assumptions about the mode the system is in and the availability or display of data are indicated for each step in the script where appropriate. Two types of or-ing for interaction options are used. One is versioning indicated by "a", "b", etc. after the step number. Thus "4a" means the "a" version of step 4, and "4b" means the "b" version of step 4. The other type of or-ing is just to put "or" between options within the same script step as in selecting a feature type or measure to perform.

Level 1

1) Access the appropriate data sources from the development environment.
2) Display, manipulate and examine data.
3) Develop image processing operators that extract evidence from imagery.
4) Extract features and measures from imagery regions.
5) Create models of objects to be recognized and measured from imagery.
6) Develop matching operators that compare regions, features and measures with segments of models.
7) Develop inference structures such as Bayesian networks or rule bases.
8) Experiment with reasoning control strategies on the inference structures.
9) Craft the user interface to yield a natural vertical application solution.
10) Test the (semi-) automated solution for robustness and reliability.

Level 2

1) Access the appropriate data sources from the development environment.
1.1a) Type-commands to access images from known locations.

Assumptions: The developer knows the image s/he wants and it is accessible by the system. The system need only have routines to use a pathname to retrieve an image. The system may need to be able to access external databases, such as a digital terrain database, and know foreign imagery formats, such as for medical images.

1.1b) Display an imagery database browser and make selections to retrieve desired imagery.

Assumptions: All display and retrieval interaction is idiot-proofed. For large retrievals, the developer is warned of the amount of data and asked to confirm the retrieval. Interface has full database capability (keys on imagery names, dates, sensor-types, general image content, etc.).

1.1c) Something in-between 1.1a (the user is smart and the system is dumb) and 1.1b (the system is smart and the user is dumb).

2) Display, manipulate and examine data
2.1) Display the selected imagery.

Assumptions: Display function is smart about window sizes versus image sizes. Display does not change aspect ratio of imagery. Clipping is optional.

2.2) Do display manipulations of imagery, at a minimum including scrolling, zooming by pixel replication, anti-aliasing rotation (e.g. the Fant warp routine), fast transpositions and re-scaling.

2.3) Text describing imagery objects or ephemeris data should be able to be moved into windows so it can be side-by-side with imagery or other signals. It should be able to be zoomed up or down or font substituted to be bigger or smaller.

2.4) Imagery should be interactively examinable for gray-level distributions and pixel values in arbitrary sub-regions of the image. In particular we should be able to look at an ascii (numerical) view of sub-regions of pixels, or get the histogram, mean and variance of pixel values in an arbitrary region. It is nice to be able to graph the profile of the pixel values under an interactively defined geo-object-line or for any arbitrary sequence of pixels. It is convenient to be able to drag a small window interactively over the image and see display of pixel values or other statistics dynamically computed and displayed.

Assumptions: These operations can be interleaved with the display and processing of any image. It should be possible to represent sub-regions for display and examination by multiple methods. At a minimum it should be possible to represent sub-regions as either regions output by connected component, or defined interactively as a polygon or other geo-object.

3) Develop image processing operators that extract evidence from imagery.

3.1) Define and move about image display windows easily and have names for them that are usable in interactive operations.

Assumptions: Image operators are smart enough to know about their need for scratchpad memory (or scratchpad windows), and whether operations can be done in place (like LUT filtering) or requires an output window (like FFT). The user is prompted or shutout from illegal operations automagically.

3.2) Do standard look-up-table (LUT) filtering. The developer should be able to easily set a LUT for gray level and/or color either by putting values in a file or an interactive data structure, or by parametrically defining a function to generate values for the LUT. Histogram equalization, parametrized gamma correction, absolute and multi-value thresholding should be available as standard LUT generating routines.

3.3) A generic method should be available to do an algorithmically parallel neighborhood operation at every pixel. The most general capability allows the developer to write any function that reads the values of the pixels in a runtime defined neighborhood (n by m, circular or hexagonal) and replaces with center pixel with a new value returned by the function. The most specific capability convolves a square fixed size neighborhood (2n+1 by 2n+1, typically with n=1,2 or 3) of each pixel with a kernel supplied by the user. Obviously, tools closer to the former are more desirable. Border processing options should be available, including constant-fill, reflection, and wrap-around (i.e. tiling). It should be optional to save results as individual images, or "pipe" them into other operators, as defined by an appropriate IP command language (oops, another solution method creeping in there...).

3.4) The method of 3.3 can be productively generalized to run along any defined geo-object. Performing neighborhood computations along the boundary of a region is a particularly useful operation for looking for gradient evidence in building models and model-matching routines.

3.5) Imagery algebra and arithmetic function operations are often performed between images; it should be easy to AND, OR or DIFF two images, and to do the operations +,-,* and / between them. Again, results should be savable as images.

3.6) A very efficient connected component capability should be available, as this routine is used often. It should be optionally 4 or 8 connected (hexagonal connectivity and "sided" connectivity options are nice too, but not as frequently used). Basic features should be computed for each component for efficiency (as they can be easy tracked during component construction, and are likely to be needed for further processing) including pixel-count, area in centimeters (if conversion constant is supplied with the imagery), number of pixels in the interior and exterior perimeters, average and variance of the gray-levels or for each of rgb, and genus (number of holes).

Assumptions: The data structures output by connected component are understood by virtually all other system components, including databases, display and browsing methods, and routines that process imagery features; see the level 2 description of step 4.

4) Extract features and measures from imagery regions.

4.1) Most features are calculated as functions using as input the numerical values and spatial relationships of a set of pixels in a connected region output by connected component. So the object-manipulating environment should make this data easy to obtain in arrays or other data structures for use in feature-computing functions.

4.2) The developer wants to create two basic classes of features. One is spatial structures, such as boundary shape descriptors and surface fits to a region, and the other is numerical (real-valued) measures of regions and their derived spatial representations, such as area, curvature, lengths, etc. Multi-resolution versions of most descriptors should be available. At any single level, the data structures representing the features at that level of resolution should be accessible by IP and feature creating routines the same way as single-resolution feature representations.

4.3) Whenever spatial representations are created, the developer wants to look at distributions of the associated measures, and to experiment with thresholding the distributions to look at various subsets of the feature space. The developer can specify the thresholding interactively as values, or can use one derived automatically from functions that look at distributions of feature values and execute criteria such as "threshold at top 5% of histogram of pixel values", of "threshold at the top 20% of high curvature points".

4.4) Search tools are used to define and process feature groupings. Feature search tools come in two basic varieties, attribute matching tools, and relational search. In attribute matching, set of regions or features are defined, typically as the output of some operator, and the developer wants to see which ones fulfill certain constraints on attributes, such as size, color, compactness, etc., stored with the feature or region. Relational search requires looking not just at each feature, but also at spatial relationships (and often other subsequent attribute comparisons) between multiple features or regions. These again come in two varieties, unordered relations, and sequential relations. Nearness is an unordered relation, but branching is an ordered, or sequential, relation. Boundary following is also a common sequential search operation.

Assumptions: All search tools understand any tools used for rapid (multi-resolution) spatial indexing of features and/or imagery.

4.5) Database storage of features is an implicit solution approach here, however if this approach is used, care must be taken to tightly integrate database access with IP and feature searching/generating routines. It may, for example, be far more efficient to pipe results from one operator to another without intermediate database storage. This could require some smarts, or option switches, to know when storage is desired. In any case, the user must understand what results are saved, which are not saved, and those that are not saved but whose process-to-create is recoverable.

5) Create models of objects to be recognized and measured from imagery.

5.1) The basic need is interactive modeling packages that can be used to create parametrized geometric object models from 1D and 2D splines, prisms, and generalized cylinders. In one scenario, a developer draws a contour on an image, extracts it and fits a1D spline to it. Another contour is extracted and spline fit; but now the ratio of of spline coefficients needs to be computed and stored, rather than forcing the absolute numerical parameters in. Relational model parametrization is of key importance; it is the ratios between parameters that get specified during the interactive sessions. Statistics governing distributions of parameters can be

interactively input, but may come from other routines (e.g. a database access of a ground truth file of imagery or feature objects).

5.2) 3D models can be represented as an ordered, linked stack of 2D models, or directly as a 3D volume, as with generalized cylinders, or, with loss of information, as a surface map of polygons in 3-space with attachments. (Of course there are other representations, but these are the three we are initially using in medical applications.) In each representation the developer wants to view (projected) instantiated models against data interactively, and to view displays of models as parameters are interactively varied.

5.3) Geometric modeling proceeds by defining the primitive model components and their spatial relationships, such as adjacency, affixments, joints, and parametric relationships between axes and surfaces. It is convenient to have a modeling language that allows specification of spatial operations as an algebra (another design note).

5.4) Constraints are now modeled between geometric parts. Displays of tables of numerical outputs and also of projected models are used to check constraint propagation between model components are parameters are varied. For example, a developer models the bones of the finger and their relationships and constraints, and then checks the relative joint flexions by varying the angle of one joint and viewing displays of the range of the other.

5.5) Population statistics may be presented as normal distributions and/or by intervals. Constraints also can exist between these, so that when one distribution is pegged at a fixed value (or small interval) based on observations, dependent distributions are modified to ranges compatible with the observation and the relationship with the first distribution. Statistics may need to be computed from a training set, typically intended to be a random sample of the population being modeled.

Assumptions: The environmental tools either are sufficient to access the training set data, or the developer has some capability to access that data.

5.6) Now that the primitive model structures are understood, full part-of and is-a hierarchical (inheritance) taxonomies are defined, and the complete structure is created.

5.7) Operators are attached to the appropriate model nodes indicating parametric relationships between the operator inputs and the model so that the operator can be machine-instantiated at runtime to gather evidence supporting or denying the presence of an instance of the model.

6) Develop matching operators that compare regions, features and measures with segments of models.

6.1) The developer experiments with interactive parameter adjustment of models, projection of 3D models into 2D predictions, and matching the predicted model against extracted features and/or regions. There are two main matching approaches. The first is identical to model instantiation: an extracted feature is fit to a model component, e.g. a set of boundary pixels are fit to a 1D spline. In the second, a sensor image acquisition model is applied to an (partly) instantiated 3D model from a hypothesized perspective. and a geo-object is created that can be matched against the geo-object implied by the region occupied by the imagery feature. This fitting procedure can be supported by an appropriately applied least-squares-fit.

Assumptions: For each model primitive, there exists (a) method(s) to match it against some type(s) of features and/or regions.

6.2) A key advantage of model-based reasoning is that constraints in instantiated parts of models can be propagated to as yet uninstantiated values of model parameters to focus predictions for further processing. Based on partial matches, the developer now exercises the prediction mechanism to see the object localization implied by the

partial match. This can be used to place values on operators, and as a guide to refining models and model matching.

Assumptions: Computation and display of predictions is easy and relatively rapid, and/or the development environment is multi-processing so that development does not have to stop and wait long for prediction results.

7) Develop inference structures such as Bayesian networks or rule bases.

7.1a) Building of the Bayes net begins by choosing the set of models that encompass the recognizable world (closed world assumption, with "other" category), and constructing a model bayes net that associates model nodes together as the competing hypotheses in Bayes nodes. (In the most general environment, there are cleverly indexed databases that allow automatic Bayes net building by building databases of conflicting models based on domain task applications.)

7.2a) Develop the conditional probability matrices between Bayes nodes based on the discriminatory evidence about the IP and other evidence gathering operators as described in step (5.7).

7.3a) Establish an initialization process that instantiates a (partial) Bayes net based on a fixed set of operations that have high probability of success (e.g. the medical hand-finder).

7.1b) Create a rulebase that captures model instantiations, evidence gathering, and decision making (about termination, for example), based on relationships between models and evidential matching results. The basic rule is of the form "If you see X, then do Y.".

Assumptions: Rule syntax and inference engines accept as inputs the results generated from evidence gathering operators, from statistical routines, and from accessing the model objects.

7.2b) Iteratively experiment with chaining in the rulebase based on alternate initialization sequences to determine both the initialization processing, and completeness of the rulebase.

8) Experiment with reasoning control strategies on the inference structures.

8.1) The developer iteratively changes the state of the Bayes net and/or rulebase or other inference structure, then manually indicates the next processing step and views the results.

Assumptions: Persistent data that the developer wants to save and reload to experiment with reasoning control includes all models, results of IP, pattern recognition, and grouping operators, matching methods, and Bayes nets and/or rulebases. It is preferable to be able to save an intermediate state in Bayes net and/or rulebase inference.

8.2) The developer runs automated processing routines such as a decision theory evaluation routine, a metarule selection strategy for multiple rule firings, or an influence diagram algorithm (that incorporates the Bayes net.)

Assumptions: Reasoning control programs accept as inputs models, inference structures and associated parameters.

8.3) The developer examines explanations of automated processing runs, and then interactively alters models, inference and/or reasoning control.

Assumptions: Explanation facilities are available for each control strategy and inference structure.

9) Craft the user interface and documentation to yield a natural vertical application solution.

9.1) Do task analysis, work studies and interface storyboards to determine the full sequence of steps, relations between domain task operations, environmental and operational constraints, resource constraints, required integration with other products and/or data, domain user technology familiarities, typical enduser occupational background and computer use capabilities (or phobias), and the uses that output information is put to.

Assumptions: The developer has access to view and analyze operational environments in the domain application area.

9.2) Rapidly prototype alternate user interfaces, and let potential end users experiment with the interface to uncover problems, design shortfalls, unexpected data dependencies, etc.

Assumptions: The developer has access to candid, knowledgable, representative and hardworking end users who have the time to evaluate the proto-interfaces.

9.3) Documentation for system is developed and tested on typical users both with and without assistance. Without assistance results are used to modify the written documentation so that assistance is unnecessary to easily run the system. With assistance results are used to test the successfulness of the actual system functioning under control of an enduser.

10) Test the (semi-) automated solution for robustness and reliability.

10.1) Determine approximately how many cases are required to establish reliability in the accuracy of the product's output measurements, then run the system over that many cases selected randomly from the population. Statistics need to be gathered.

10.2) Test that each system component has correctly implemented the required algorithms.

10.3) Establish software metrics for system reliability in terms of continuous functioning, and test the system accordingly. Again statistics are gathered, including timing and memory usage.

9.0 References

[Edelson et.al., 88] Edelson, D., J. Dye, T. Esselman, M. Black, and C. McConnell, "VIEW Programmer's Manual", Advanced Decision Systems, Mountain View, California, June 1988.

[KBVision, 87] "KBVision Programmer's Manual", Amerinex Artificial Intelligence, Amherst, Mass., 1987.

[Lawton and Levitt, 89] Lawton D.T. and T.S. Levitt, "Knowledge-Based Vision for Terrestrial Robots", Proc. DARPA IU Workshop, Morgan Kauffman, San Mateo, California, May, 1989. pp 128-133.

[Lawton and McConnell, 88] Lawton, D.T. and C.C. McConnell, "Image Understanding Environments", Proc. IEEE, Vol. 76, No. 8, August, 1988, pp. 1036-1050.

[McConnell et. al., 88] McConnell, C., K.Riley, and D. Lawton, "Powervision Manual", Advanced Decision Systems, Mountain View, California, 1988.

[Quam, 84] Quam, L., "The Image Calc Vision System", Stanford Research Institute, Menlo Park, California, 1984.

[Waltzman, 90] Unpublished notes at the Image Understanding Environment Requirements Meeting, Hughes AI Laboratory, Malibu, California, May, 1990.

Appendix A Object Hierarchy

- Container
  - Point
    - Point1d
    - Point2d
    - Point3d
    - PointNd
  - Curve
    - Curve1d
      - Signal
    - Curve2d
      - SimpleCurve2d
        - PointCurve2d
        - EdgeCurve2d
        - PolynomialCurve2d
        - SplineCurve2d
          - BezierCurve2d
    - Curve3d
      - SimpleCurve3d
        - PointCurve3d
        - EdgeCurve3d
        - PolynomialCurve3d
        - SplineCurve3d
          - BezierCurve3d
    - CurveNd
      - SimpleCurveNd
        - PointCurveNd
        - EdgeCurveNd
        - PolynomialCurveNd
        - SplineCurveNd
          - BezierCurveNd
  - Surface
    - Surface2d
      - SimpleSurface2d
        - ConstantSurface2d
          - Box
          - Polygon
            - Parallelogram
          - RLE
        - ValuedSurface2d
          - Image
          - PolygonImage
            - WarpedImage
            - TiltedImage
          - RLE_Image
    - ConnectedSurface2d
      - AggregateSurface2d

```
            Surface3d
                  SimpleSurface3d
                        ConstantSurface3d
                        ValuedSurface3d
                  ConnectedSurface3d
                  AggregateSurface3d
            SurfaceNd
                  SimpleSurfaceNd
                        ConstantSurfaceNd
                        ValuedSurfaceNd
                  ConnectedSurfaceNd
                  AggregateSurfaceNd


      Solid
            Solid3d
                  SimpleSolid3d
                        ConstantSolid3d
                              GeneralizedCylinder
                              CSG's
                        ValuedSolid3d
                              Space
                              BoundedSpace
                  ConnectedSolid3d
                  AggregateSolid3d
            SolidNd
                  SimpleSolidNd
                        ConstantSolidNd
                        ValuedSolidNd
                  ConnectedSolidNd
                  AggregateSolidNd

      HyperSolid
            HyperSolidNd
                  SimpleHyperSolidNd
                        ConstantHyperSolidNd
                        ValuedHyperSolidNd
                              HyperSpace
                              BoundedHyperSpace
                  ConnectedHyperSolidNd
                  AggregateHyperSolidNd

Coordinate
      Global
      Local
            Base


Collection  (ContainerParts)

      Array
            ByteArray
            Array2d
                  ByteArray2d
```

```
        Array3d
                ByteArray3d
        ArrayNd
                ByteArrayNd

Stream
        ByteStream
        Stream2d
                ByteStream2d
        Stream3d
                ByteStream3d
        StreamNd
                ByteStreamNd

Graph
        Tree
                List

Record
```

Appendix B Image Processing Source Libraries

ALV Toolkit

Contact: alv-users-request@uk.ac.bris.cs

Description:

Public domain image processing toolkit written by Phill Everson (everson@uk.ac.bris.cs). Supports the following:

- image display
- histogram display
- histogram equalization
- thresholding
- image printing
- image inversion
- linear convolution
- 27 programs, mostly data manipulation

BUZZ

Contact:

Tehnical:	Licensing:
John Gilmore	Patricia Altman
(404) 894-3560	(404) 894-3559

Artificial Intelligence Branch
Georgia Tech Research Institute
Georgia Institute of Technology
Atlanta, GA 30332

Description:

BUZZ is a comprehensive image processing system developed at Georgia Tech. Written in VAX FORTRAN (semi-ported to SUN FORTRAN), BUZZ includes algorithms for the following:

- image enhancement
- image segmentation
- feature extraction
- classification

HIPS

Contact: SharpImage Software
P.O. Box 373
Prince St. Station
NY, NY 10012

Michael Landy (212) 998-7857
landy@nyu.nyu.edu

Description:

HIPS consists of general UNIX pipes that implement image processing operators. They can be chained together to implement more complex operators. Each image stores history of transformations applied. HIPS is available, along with source code, for a $3000 one-time license fee.

HIPS supports the following:

- simple image transformations
- filtering
- convolution
- Fourier and other transforms
- edge detection and line drawing manipulation
- image compression and transmission
- noise generation
- image pyramids
- image statistics
- library of convolution masks
- 150 programs in all

LABO IMAGE

Contact: Thierry Pun Alain Jacot-Descombes
+(4122) 87 65 82 +(4122) 87 65 84
pun@cui.unige.ch jacot@cuisun.unige.ch

Computer Science Center
University of Geneva
12 rue du Lac
CH-1207
Geneva, Switzerland

Description:

Interactive window based software for image processing and analysis. Written in C. Source code available. Unavailable for use in for-profit endeavours. Supports the following:

- image I/O
- image display
- color table manipulations
- elementary interactive operations:
  - region outlining
  - statistics
  - histogram computation

- elementary operations:
  - histogramming
  - conversions
  - arithmetic
  - images and noise generation
- interpolation: rotation/scaling/translation
- preprocessing: background subtraction, filters, etc;
- convolution/correlation with masks, image; padding
- edge extractions
- region segmentation
- transforms: Fourier, Haar, etc.
- binary mathematical morphology, some grey-level morphology
- expert-system for novice users
- macro definitions, save and replay

Support for storage to disk of the following:
- images
- vectors (histograms, luts)
- graphs
- strings

NASA IP Packages

VICAR
ELAS -- Earth Resources Laboratory Applications Software
LAS -- Land Analysis System

Contact: COSMIC (NASA Facility at Georgia Tech)
Computer Center
112 Barrow Hall
University of Georgia
Athens, GA 30601
(404) 542-3265

Description:

VICAR, ELAS, and LAS are all image processing packages available from COSMIC, a NASA center associated with Georgia Tech. COSMIC makes reusable code available for a nominal license fee (i.e. $3000 for a 10 year VICAR license).

VICAR is an image processing package written in FORTRAN with the following capability:

- image generation
- point operations
- algebraic operations
- local operations
- image measurement
- annotation and display

- geometric transformation
- rotation and magnification
- image combination
- map projection
- correlation and convolution
- fourier transforms
- stereometry programs

"ELAS was originally developed to process Landsat satellite data, ELAS has been modified over the years to handle a broad range of digital images, and is now finding widespread application in the medical imaging field ... available for the DEC VAX, the CONCURRENT, and for the UNIX environment." -- from NASA Tech Briefs, Dec. 89

"... LAS provides a flexible framework for algorithm development and the processing and analysis of image data. Over 500,000 lines of code enable image repair, clustering, classification, film processing, geometric registration, radiometric correction, and manipulation of image statistics." -- from NASA Tech Briefs, Dec. 89

OBVIUS

Contact: for ftp --> whitechapel.media.mit.edu
otherwise --> heeger@media-lab.media.mit.edu
MIT Media Lab Vision Science Group
(617) 253-0611

Description:

OBVIUS is an object-oriented visual programming language with somesupport for imaging operations. It is public domain CLOS/LISP software. It supports a flexible user interface for working with images. It provides a library of image processing routines:

- point operations
- image statistics
- convolutions
- fourier transforms

POPI (DIGITAL DARKROOM)

Contact: Rich Burridge
richb@sunaus.sun.oz.AU
-- or --
available for anonymous ftp from ads.com
(pub/VISION-LIST-BACKISSUES/SYSTEMS)

Description:

Popi was originally written by Gerard J. Holzmann - AT&T Bell Labs.

This version is based on the code in his Prentice Hall book, "Beyond Photography - the digital darkroom," ISBN 0-13-074410-7, which is copyright (c) 1988 by Bell Telephone Laboratories, Inc.

VIEW (Lawrence Livermore National Laboratory)

Contact: Fran Karmatz
Lawrence Livermore National Laboratory
P.O. Box 5504
Livermore, CA 94550
(415) 422-6578

Description:

Window-based image-processing package with on-line help and user manual. Multidimensional (2 and 3d) processing operations include:

- image display and enhancement
- pseudocolor
- point and neighborhood operations
- digital filtering
- fft
- simulation operations
- database management
- sequence and macro processing

Written in C and FORTRAN, source code included. Handles multiple dimensions and data types. Available on Vax, Sun 3, and MacII.